o

Query Exercises

DANIELE APILETTI

POLITECNICO DI TORINO

Data Model

Given the following collection of books

```
{_id:ObjectId("5fb29ae15b99900c3fa24292"),
 title:”MongoDb Guide",
 tag:[”mongodb”,”guide”,“database”],
 n:100,
 review_score:4.3,
 price:[{v: 19.99, c: “€”,  country: “IT”},
        {v: 18, c: “£”, country:”UK”} ],
 author: {_id: 1,
          name:”Mario”,
          surname: “Rossi”}
},
{_id:ObjectId("5fb29b175b99900c3fa24293",
 title:”Developing with Python",
 tag:[”python”,”guide”,“programming”],
 n:352,
 review_score:4.6,
 price:[{v: 24.99, c: “€”,  country: “IT”},
        {v: 19.49, c: “£”, country:”UK”} ],
 author: {_id: 2,
          name:”John”,
          surname: “Black”}
}, …
```

price currency

price value

number of pages

Exercises

1. Find all the books with a **number of pages** greater than 250
2. Find all the books **authored** by Mario Rossi
3. Find all the books with a **price** less than 20 € for **Italy** (IT)

Solutions

- Find all the books with a number of pages greater than 250

```
db.book.find({n: {$gt: 250 }})
```

- Find all the books authored by Mario Rossi

```
db.book.find({"author.name": "Mario", "author.surname": "Rossi" })
```

- Find all the books with a price less than 20 € for the country Italy (IT)

```
db.book.find({"price":  {$elemMatch: {"v": {$lt: 20}, "country": "IT" }}} )
```

Data Model

Given the following collection of books

```
{_id:ObjectId("5fb29ae15b99900c3fa24292"),
 title:”MongoDb Guide",
 tag:[”mongodb”,”guide”,“database”],
 n:100,
 review_score:4.3,
 price:[{v: 19.99, c: “€”,  country: “IT”},
        {v: 18, c: “£”, country:”UK”} ],
 author: {_id: 1,
          name:”Mario”,
          surname: “Rossi”}
},
{_id:ObjectId("5fb29b175b99900c3fa24293",
 title:”Developing with Python",
 tag:[”python”,”guide”,“programming”],
 n:352,
 review_score:4.6,
 price:[{v: 24.99, c: “€”,  country: “IT”},
        {v: 19.49, c: “£”, country:”UK”} ],
 author: {_id: 2,
          name:”John”,
          surname: “Black”}
}, …
```

price currency

price value

number of pages

Exercises

1. Increase the **review score** of 0.2 points

   for all the books with the **tag** "database"

2. Insert the **tag** "NoSQL" for all the books with **tag** "mongodb"

3. Insert the **publisher** for all the documents **authored** by Mario Rossi with the default value {'name': 'Polito', city:'Turin'}

Solutions

- Increase the review score of 0.1 for all the books with the tag database

```
db.book.updateMany({tag: "database" },  { $inc: {review_score: 0.2} })
```

- Insert the tag "NoSQL" for all the books with tag "mongodb"

```
db.book.updateMany({tag: "mongodb" },  { $addToSet: {tag: "NoSQL"} })
```

- Insert the publisher for all the documents authored by Mario Rossi with the default value {'name': 'Polito', city:'Turin'}

```
db.book.updateMany(
 {"author.name": "Mario", "author.surname": "Rossi"},
 {$set: {publisher: {name:"Polito", city:"Turin"}}} )
```

Data Model

Given the following collection of books

```
{_id:ObjectId("5fb29ae15b99900c3fa24292"),
 title:”MongoDb Guide",
 tag:[”mongodb”,”guide”,“database”],
 n:100,
 review_score:4.3,
 price:[{v: 19.99, c: “€”,  country: “IT”},
        {v: 18, c: “£”, country:”UK”} ],
 author: {_id: 1,
          name:”Mario”,
          surname: “Rossi”}
},
{_id:ObjectId("5fb29b175b99900c3fa24293",
 title:”Developing with Python",
 tag:[”python”,”guide”,“programming”],
 n:352,
 review_score:4.6,
 price:[{v: 24.99, c: “€”,  country: “IT”},
        {v: 19.49, c: “£”, country:”UK”} ],
 author: {_id: 2,
          name:”John”,
          surname: “Black”}
}, …
```

price currency

price value

number of pages

Exercises

1. Find the maximum, the minum and the average **price** of all the books with **tag** “database”
2. Compute the number of books **authored** by Mario Rossi

Solutions

- Find the maximum, the minum and the average price of all the books with tag "database"

```
db.book.aggregate([ {$match: {tag: "database" }},
                    {$unwind: "$price"},
                    {$group: {_id: null,
                        avg: {$avg: "$price.v"} ,
                        min: {$min: "$price.v"} ,
                        max: {$max: "$price.v"} } } ])
```

- Compute the number of books authored by Mario Rossi

```
db.book.count({ "author.name": "Mario", "author.surname": "Rossi" })
```

```
db.book.find({ "author.name": "Mario", "author.surname": "Rossi" }).count()
```

Politecnico di Torino

DBMG

Design patterns (1)

DANIELE APILETTI

POLITECNICO DI TORINO

Your responsibility: a **flexible** schema

- Unlike SQL databases, **collections** do **not** require its **documents** to have the **same schema**, i.e., the following properties might change:
  - the **set of fields** and
  - the **data type** for the same field
- In practice, however, documents in a collection share a similar structure
  - Which is the best **document structure**?
  - Are there **patterns** to address common applications?
- It is possible to enforce **document validation** rules for a collection during update and insert operations

Example: Embedded vs reference

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

Atomicity of Write Operations

- A write operation is atomic on the level of **a single document**, even if the operation modifies multiple embedded documents *within* a single document
- When a single write operation (e.g. *db.collection.updateMany()*) modifies multiple documents, the modification of **each document is atomic**, but the operation as a whole is **not atomic**
- For situations requiring atomicity of reads and writes to multiple documents (in a single or multiple collections), MongoDB supports **multi-document transactions**:
  - **in version 4.0**, MongoDB supports multi-document transactions on replica sets
  - **in version 4.2**, MongoDB introduces distributed transactions, which adds support for multi-document transactions on sharded clusters and incorporates the existing support for multi-document transactions on replica sets

Schema validation

MongoDB can perform schema validation during updates and insertions. Existing documents do not undergo validation checks until modification.

- *validator*: specify validation **rules or expressions** for the **collection**
- *validationLevel*: determines how strictly MongoDB applies validation rules to existing documents during an update
  - ***strict***, the default, applies to all changes to any document of the collection
  - ***moderate***, applies only to existing documents that already fulfill the validation criteria or to inserts
- *validationAction*: determines whether MongoDB should **raise error and reject** documents that violate the validation rules or **warn** about the violations in the log but allow invalid documents

```
db.createCollection( <name>,
                {validator: <document>,
                validationLevel: <string>,
                validationAction: <string>,
                } )
```

JSON Schema validator

- Starting in version 3.6, MongoDB supports JSON Schema validation (recommended)
- To specify JSON Schema validation, use the *$jsonSchema* operator

```
db.createCollection("students",
                { validator: {
                                $jsonSchema: {
                                                bsonType: "object",
                                                required: [ "name", "year" ],
                                                properties: {
                                                                name: {
                                                                                bsonType: "string",
                                                                                description: "must be a string and is required"
                                                                },
                                                                year: {
                                                                                bsonType: "int",
                                                                                minimum: 2000,
                                                                                maximum: 2099,
                                                                                description: "must be an integer in [2000, 2099] and is required»
                                                                }
                                                }
                                }
                }
})
```

Query Expression schema validator

In addition to JSON Schema validation that uses the $jsonSchema query operator, MongoDB supports validation with **other query operators**, except for:

- *$near*, *$nearSphere*, *$text*, and *$where* operators
- Note: users can bypass document validation with *bypassDocumentValidation* option.

```
db.createCollection( "contacts",
            { validator: {
                        $or: [
                                    { phone: { $type: "string" } },
                                    { email: { $regex: /@mongodb\.com$/ } },
                                    { status: { $in: [ "Unknown", "Incomplete" ] } }
                        ]
            }
} )
```

Designing **factors**

- Atomicity
  - Embedded Data Model vs Multi-Document Transaction
- Sharding
  - selecting the proper shard key has significant implications for performance, and can enable or prevent query isolation and increased write capacity
- Indexes
  - each index requires at least 8 kB of data space.
  - adding an index has some negative performance impact for write operations
  - collections with high read-to-write ratio often benefit from additional indexes
  - when active, each index consumes disk space and memory
- Data Lifecycle Management
  - the Time to Live feature of collections expires documents after a period of time

Building with patterns

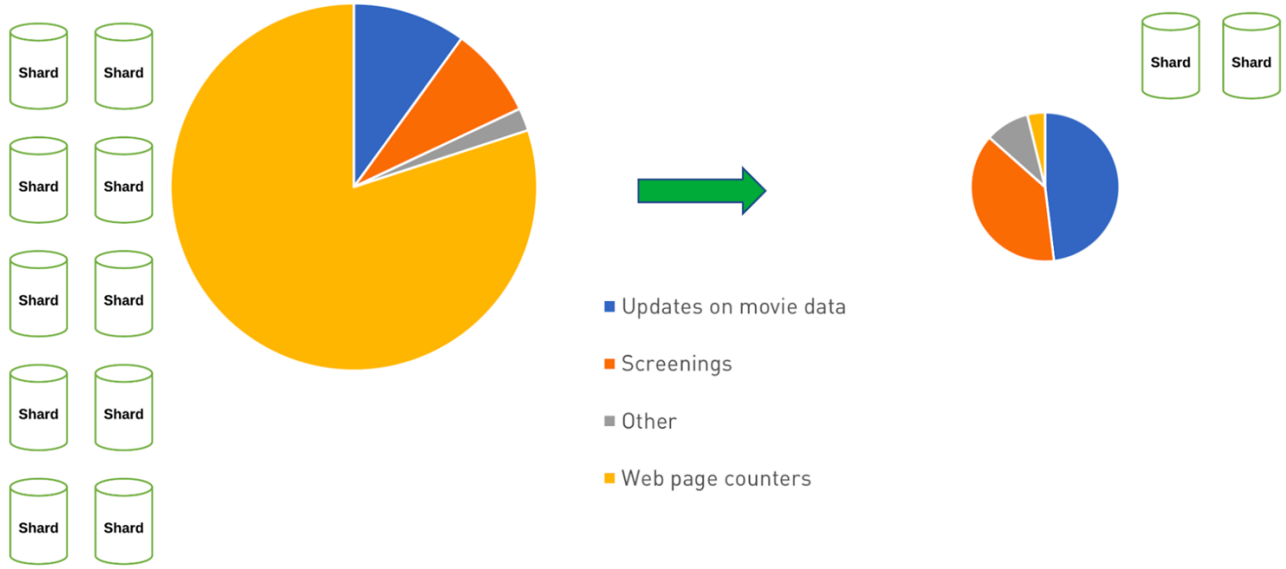
1. Approximation
2. Attribute
3. Bucket
4. Computed
5. Document Versioning
6. Extended Reference
7. Outlier
8. Pre-allocation
9. Polymorphic
10. Schema Versioning
11. Subset
12. Tree

"a driving force in what your **schema** should look like, is what the **data access patterns** for that data are"

source: https://www.mongodb.com/blog/post/building-with-patterns-the-extended-reference-pattern

1) Approximation

- Let's say that our city planning strategy is based on needing one fire engine per 10,000 people.
- instead of updating the population in the database with **every change**, we could build in a counter and only update by 100, 1% of the time.
- Another option might be to have a function that returns a random number. If, for example, that function returns a number from 0 to 100, it will return 0 around 1% of the time. When that condition is met, we increase the counter by 100.
- Our **writes are significantly reduced** here, in this example by 99%.
- when working with large amounts of data, the impact on performance of write operations is large too.



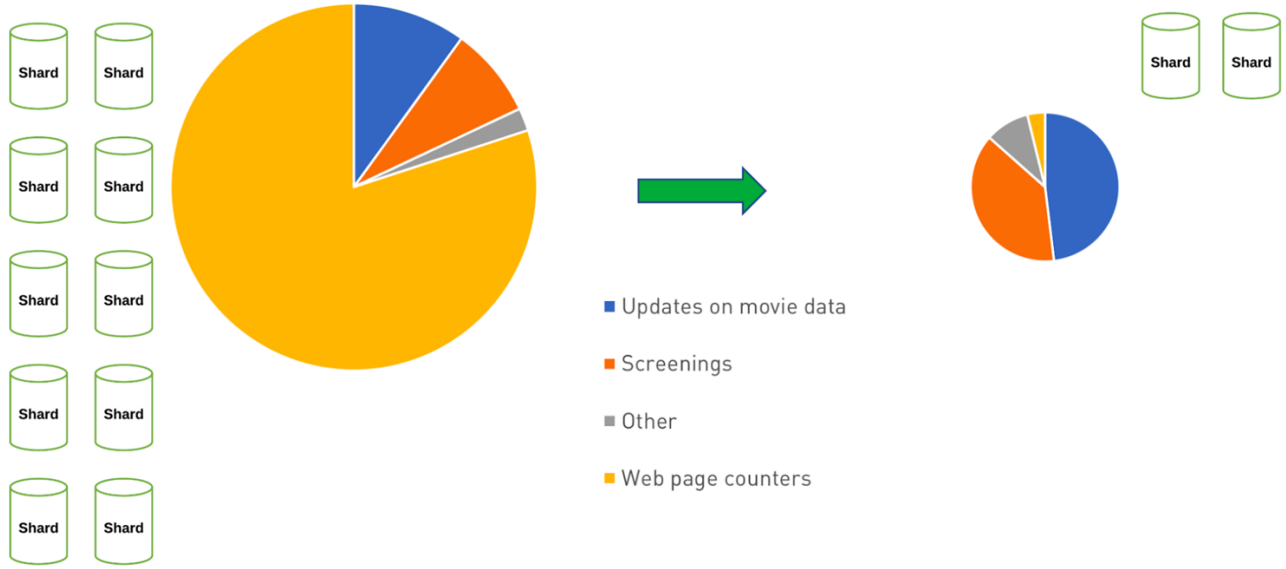
Examples

- **population counter**
- **movie website counter**

source: https://www.mongodb.com/blog/post/building-with-patterns-the-approximation-pattern

1) Approximation

- Useful when
  - expensive calculations are frequently done
  - the precision of those calculations is not the highest priority
- Pros
  - fewer writes to the database
  - no schema change required
- Cons
  - exact numbers aren't being represented
  - implementation must be done in the application



Examples

- **population counter**
- **movie website counter**

source: https://www.mongodb.com/blog/post/building-with-patterns-the-approximation-pattern

2) Attribute

- Let's think about a collection of **movies**.
- The documents will likely have similar fields involved across all the documents:
  - title, director, producer, cast, etc.
- Let's say we want to search on the **release date**: which release date? Movies are often released on different dates in different countries.
- A search for a release date will require looking across **many fields** at once, we'd need **several indexes** on our movies collection.

```
{
    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"),
    ...
}
```

- Move this subset of information into an array and reduce the indexing needs. We turn this information into an **array of key-value pairs**

```
"specs": [
    { k: "volume", v: "500", u: "ml" },
    { k: "volume", v: "12", u: "ounces" }
]
```

2) Attribute

- Useful when
  - there is a subset of fields that share common characteristics
  - the fields we need to sort on are only found in a small subset of documents
- Pros
  - fewer indexes are needed, e.g., {"releases.location": 1, "releases.date": 1}
  - queries become simpler to write and are generally faster
- Example
  - **product catalog**

Source: https://www.mongodb.com/blog/post/building-with-patterns-the-attribute-pattern

```
{
    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")
        },
        …
    ],
    …
}
```

3) Bucket

- With data coming in as a stream over a period of time (time series data) we may be inclined to store **each measurement** in **its own document,** as if we were using a relational database.
- We could end up having to **index** *sensor_id* and *timestamp* for every single measurement to enable rapid access.
- We can **"bucket" this data, by time,** into documents that hold the measurements from a particular **time span**.
- We can also programmatically add **additional information** to each of these "buckets".
- Benefits in terms of index size savings, potential query simplification, and the ability to use that **pre-aggregated data** in our documents.

```
{
   sensor_id: 12345,
   timestamp: ISODate("2019-01-31T10:00:00.000Z"),
   temperature: 40
}

{
   sensor_id: 12345,
   timestamp: ISODate("2019-01-31T10:01:00.000Z"),
   temperature: 40
}

{
   sensor_id: 12345,
   timestamp: ISODate("2019-01-31T10:02:00.000Z"),
   temperature: 41
}
```

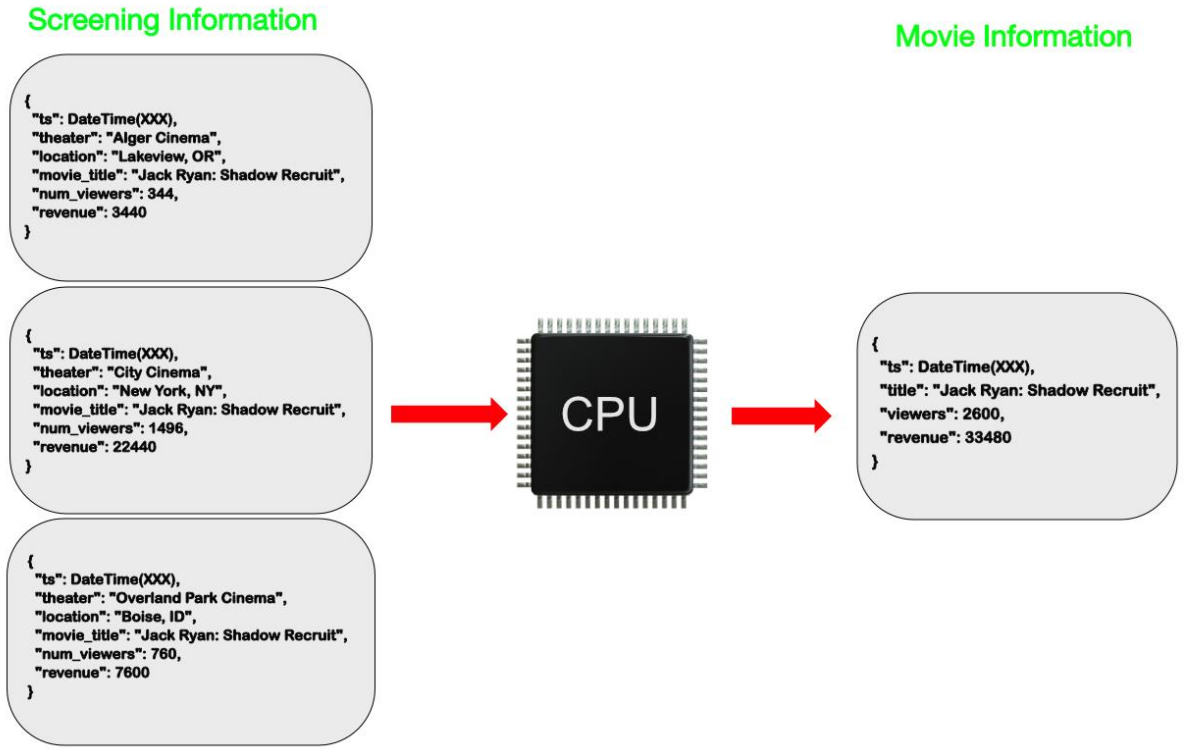
3) Bucket

- Useful when
  - needing to manage streaming data
  - time-series
  - real-time analytics
  - Internet of Things (IoT)
- Pros
  - reduces the overall number of documents in a collection
  - improves index performance
  - can simplify data access by leveraging pre-aggregation, e.g., average temperature = sum/count
- Examples
  - **IoT, time series**

```
{
    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
}
```

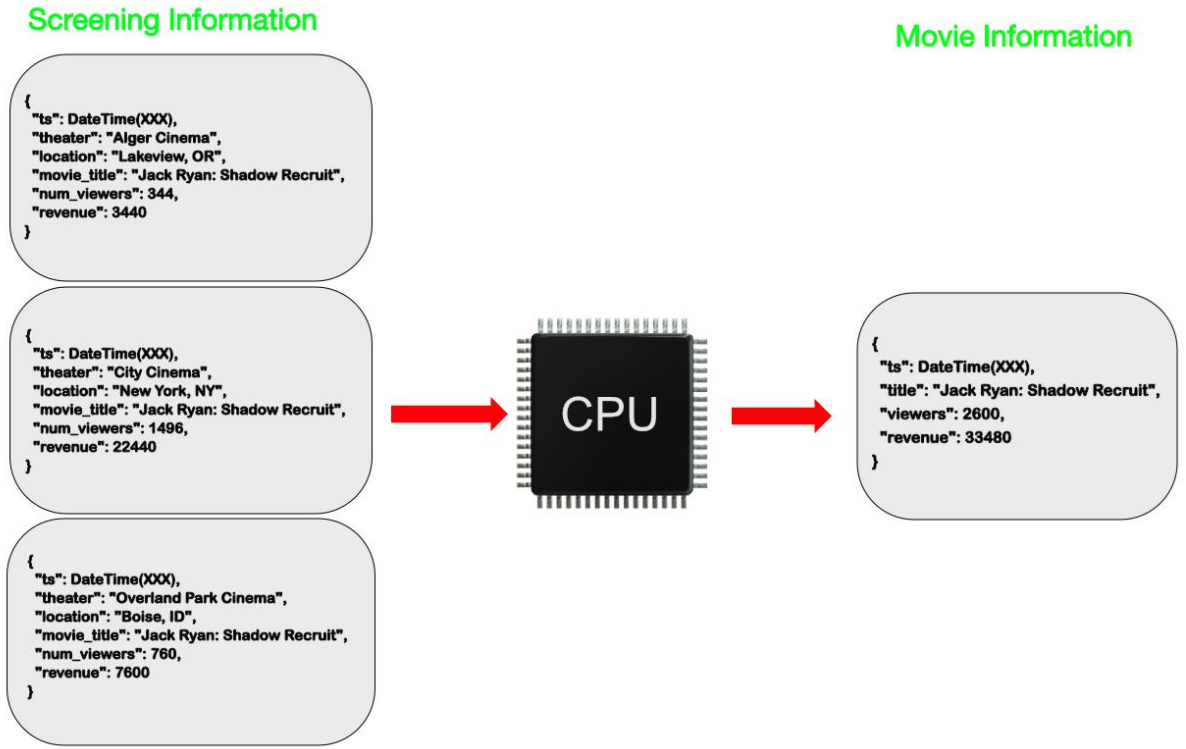
4) Computed

- The usefulness of data becomes much more apparent when we can compute values from it.
  - What's the total sales revenue of …?
  - How many viewers watched …?
- These types of questions can be answered from data stored in a database but must be computed.
- Running these **computations each time,** they're requested though becomes a highly resource-intensive process, especially on huge datasets.
- Example: a movie review website, every time we visit a movie webpage, it provides information about the number of cinemas the movie has played in, the total number of people who've watched the movie, and the overall revenue.



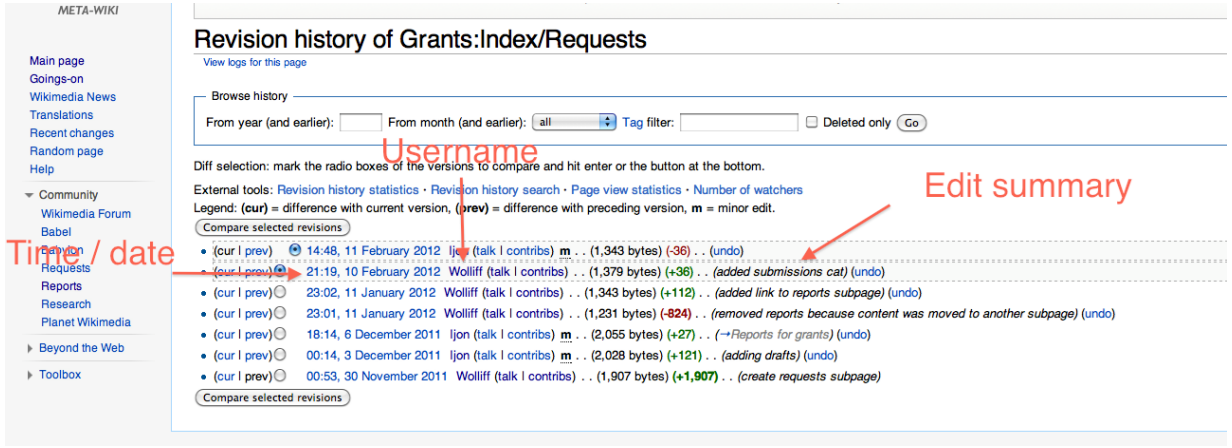
4) Computed

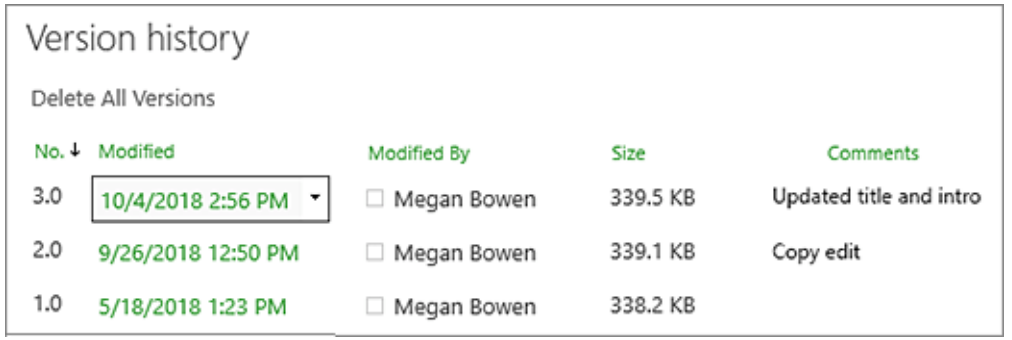
- Useful when
  - very read-intensive data access patterns
  - data needs to be repeatedly computed by the application
  - computation done in conjunction with any update or at defined intervals - every hour for example
- Pros
  - reduction in CPU workload for frequent computations
- Cons
  - it may be difficult to identify the need for this pattern
- Examples
  - **revenue or viewer**
  - **time series data**
  - **product catalogs**



5) Document Versioning

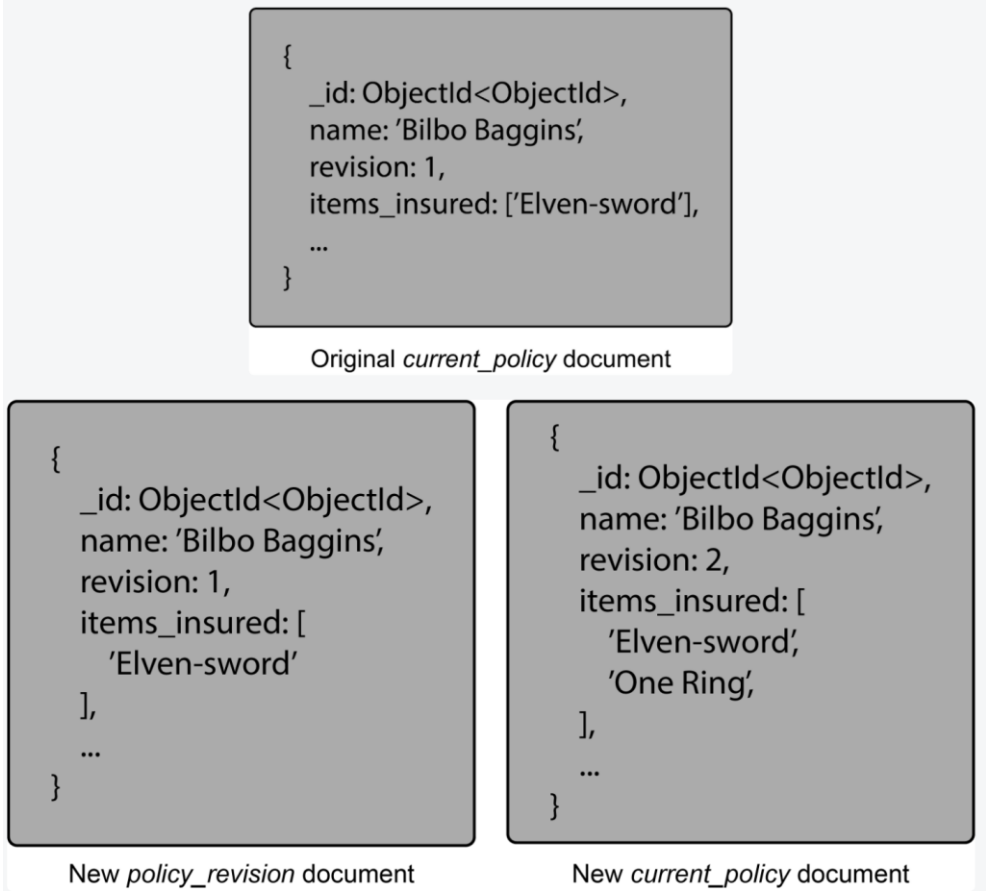
- In most cases we query **only the latest** state of the data.
  - What about situations in which we need to query previous states of the data?
  - What if we need to have some functionality of version control of our documents?
- Goal: keeping the version history of documents available and usable
- Assumptions about the data in the database and the data access patterns that the application makes
  - Limited number of **revisions**
  - Limited number of versioned **documents**
  - Most of the queries performed are done on the **most recent** version of the document





5) Document Versioning

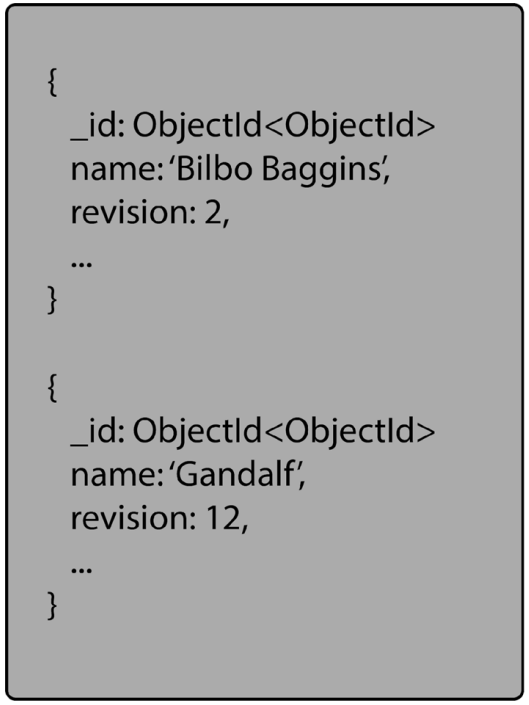
- An insurance company might make use of this pattern.
  - Each customer has a "standard" policy and a second portion that is specific to that customer.
  - This second portion would contain a **list of policy add-ons** and a list of specific items that are being insured.
- As the customer changes which specific items are insured, this information needs to be updated while the historical information needs to be available as well.
- When a customer purchases a new item and wants it added to their policy, a new *policy_revision* document is created using the *current_policy* document.
- A *version* field in the document is then incremented to identify it as the latest **revision** and the customer's changes added.



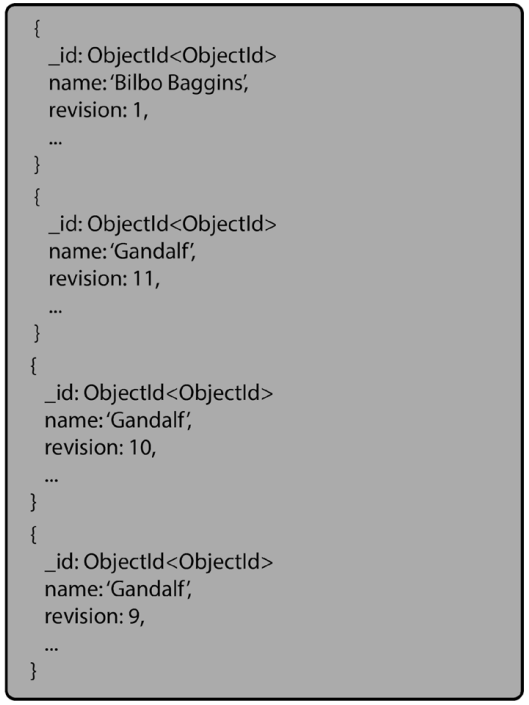
5) Document Versioning

The newest revision will be stored in the ***current_policies*** **collection** and the old version will be written to the ***policy_revisions*** **collection**.

- Pros
  - easy to implement, even on existing systems
  - no performance impact on queries on the latest revision
- Cons
  - doubles the number of writes
  - queries need to target the correct collection
- Examples
  - **financial industries**
  - **healthcare industries**



current_policies collection



policy_revisions collection

source: https://www.mongodb.com/blog/post/building-with-patterns-the-document-versioning-pattern

6) Extended Reference

In an e-commerce application

- the order
- the customer
- the inventory

are separate logical entities

Order Collection

```
{
 _id: ObjectId("507f1f77bcf86cd799439011"),
 date: ISODate("2019-02-18"),
 customer_id: 123,
 order: [
  {
     product: "widget",
     qty: 5,
     cost: {
        value: NumberDecimal("11.99"),
        currency: "USD"
     }
  }
 ]
}
```

Customer Collection

```
{
    _id: 123,
    name: "Katrina Pope",
    street: "123 Main St",
    city: "Somewhere",
    country: "Someplace",
    ...
}
```

Inventory Collection

```
{
    _id: ObjectId("507f1f77bcf86cd111111111"),
    name: "widget",
    cost: {
       value: NumberDecimal("11.99"),
       currency: "USD"
    },
    on_hand: 98325,
    ...
}
```

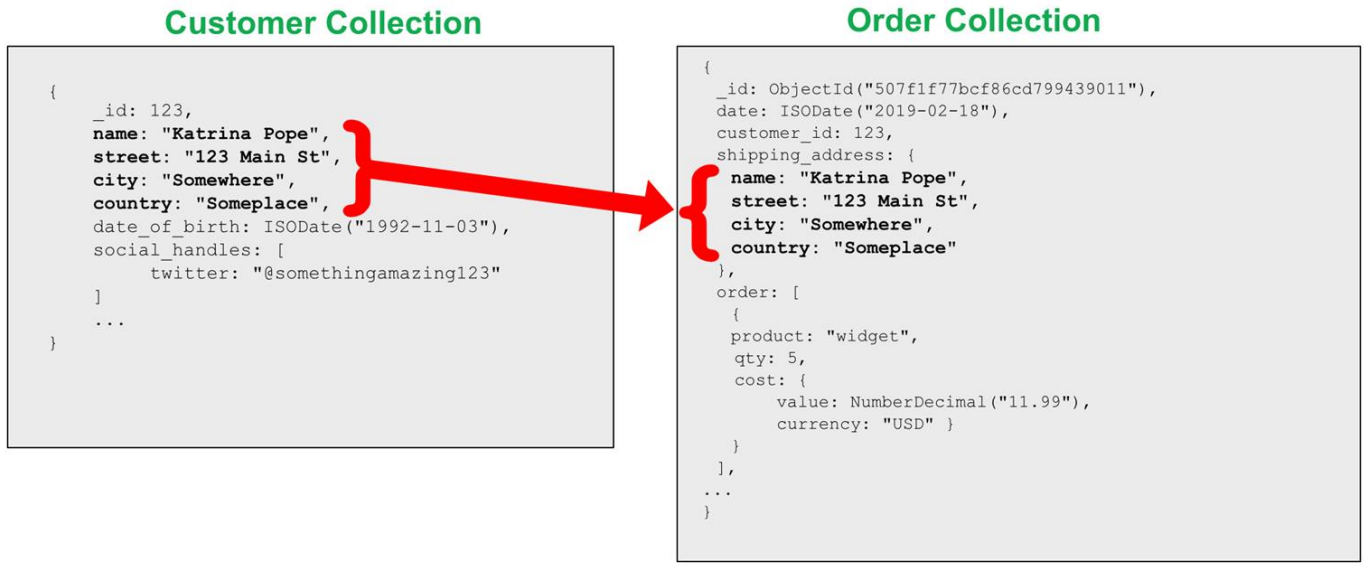
- However, the full retrieval of an order requires to **join** data from different entities
- A customer can have N orders, creating a 1-N relationship
- Embedding all the customer information inside each order
  - avoids the JOIN operation
  - results in a lot of **duplicated** information
  - not all the customer data may be actually needed

6) Extended Reference

Instead of **embedding** (i.e., duplicating) **all** the data of an external entity (i.e., another document), we only copy the fields we access frequently.

Instead of including a **reference** to join the information, we only embed those fields of the highest priority and most frequently accessed.

- Useful when
  - your application is experiencing lots of JOIN operations to bring together frequently accessed data
- Pros
  - improves performance when there are a lot of **join** operations
  - faster reads and a reduction in the complexity of data fetching
- Cons
  - data **duplication**, it works best if such data rarely change (e.g., user-id, name)
  - Sometimes duplication of data is better because you keep the **historical values** (e.g., shipping address of the order)



Acknowledgment

Bibliography

For further information on the content of these slides, please refer to the book

"Design with MongoDB"

Best Models for Applications

by Alessandro Fiori

https://flowygo.com/en/projects/design-with-mongodb/

Politecnico di Torino

DBMG

Design patterns (2)

DANIELE APILETTI

POLITECNICO DI TORINO

Building with patterns

- Does your application do more **reads** than **writes**?
- **Which pieces** of data need to be together when read from the database?
- What **performance** considerations are there?
- **How large** are the documents?
- How large will they get?
- How do you anticipate your data will **grow** and **scale**?

"a driving force in what your **schema** should look like, is what the **data access patterns** for that data are"

source: https://developer.mongodb.com/how-to/polymorphic-pattern

Building with patterns

1. Approximation
2. Attribute
3. Bucket
4. Computed
5. Document Versioning
6. Extended Reference
7. Outlier
8. Pre-allocation
9. Polymorphic
10. Schema Versioning
11. Subset
12. Tree

"MongoDB is **schema-less**. In fact, schema design is very important in MongoDB.
The hard fact is that most performance issues we've found trace back to poor **schema design**."

"When thinking of schema design, we should be thinking of **performance**, scalability, and **simplicity**."

source: https://developer.mongodb.com/how-to/polymorphic-pattern

7) Outlier

- E-Commerce selling books
  - who has purchased a particular book?
  - store an array of *user_id* who purchased the book, in each book document
- You have a solution that works for 99.99% of the cases, but what happens when a top-seller book is released?
  - You cannot store millions of user_ids due to the document size limit (16 Mbyte)
- Totally **redesigning for the outlier** is detrimental for the typical conditions
  - The outlier pattern prevents a few queries or documents from driving our solution towards one that would not be optimal for **the majority of our use cases**

```
{
    "_id": ObjectID("507f191e810c19729de860ea"),
    "title": "Harry Potter, the Next Chapter",
    "author": "J.K. Rowling",
    …,
   "customers_purchased": ["user00", "user01", "user02", …, "user999"],
   "has_extras": "true"
}
```

- Add a new field to "flag" the document as an outlier, e.g., "*has_extras*"
- Move the overflow information into a separate document linked with the book's id.
- Inside the application, we would be able to easily determine if a document "has extras".
- Only in such outlier cases, the application would retrieve the extra information.

7) Outlier

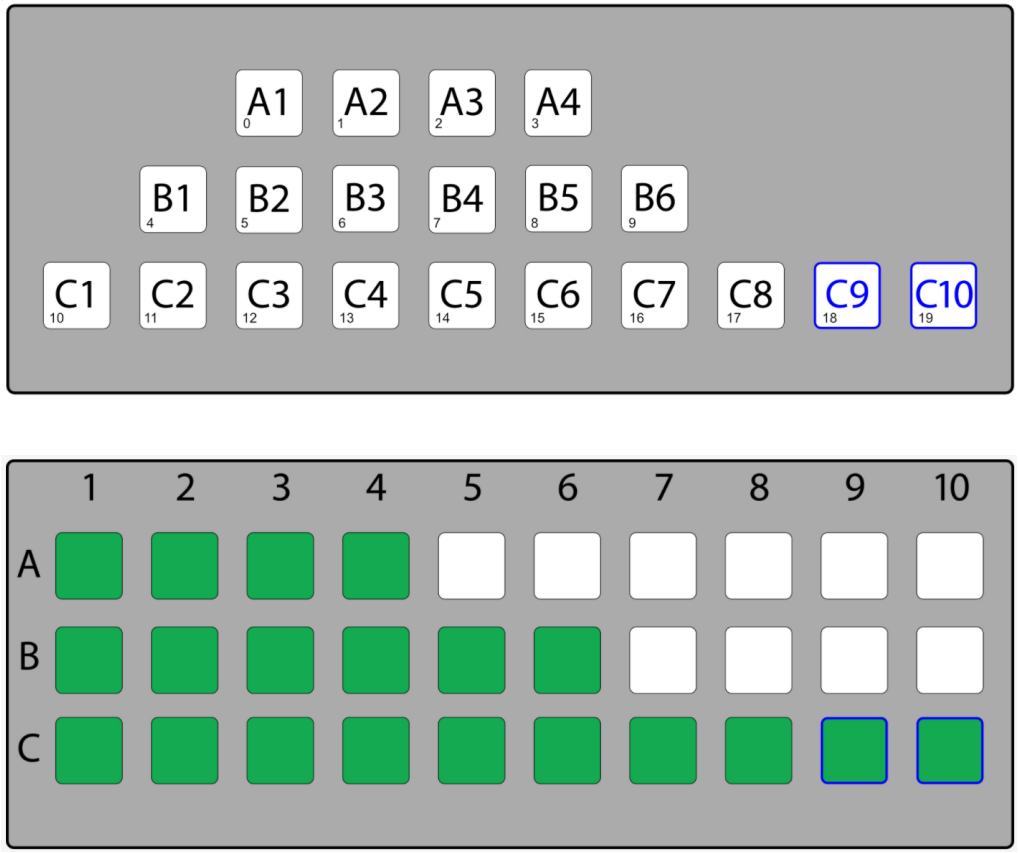
- Useful when
  - few queries or documents that don't fit into the rest of your typical data patterns
- Pros
  - prevents a few documents or queries from determining an application's solution.
  - queries are tailored for "typical" use cases, but outliers are still addressed
- Cons
  - often tailored for specific queries, therefore ad hoc queries may not perform well
  - much of this pattern is done with application code
- Examples
  - **social network relationships**
  - **book sales**
  - **movie reviews**

```
{
    "_id": ObjectID("507f191e810c19729de860ea"),
    "title": "Harry Potter, the Next Chapter",
    "author": "J.K. Rowling",
    …,
   "customers_purchased": ["user00", "user01", "user02", …, "user999"],
   "has_extras": "true"
}
```

- Add a new field to "flag" the document as an outlier, e.g., "*has_extras*"
- Move the overflow information into a separate document linked with the book's id.
- Inside the application, we would be able to easily determine if a document "has extras".
- Only in such outlier cases, the application would retrieve the extra information.

8) Pre-allocation

- Represent **a theater room** as a 2-dimensional array where each seat has a "row" and "number", for example, the seat "C7"
- Some rows may have fewer seats, however finding the seat "B3" is faster and cleaner in a **2-dimensional array**, than having a complicated formula to find a seat in a one-dimensional array that has only cells for the existing seats.
- Being able to identify accessible seating is also easier as a **separate array** can be created for those seats.



8) Pre-allocation

Another example: a reservation system where a resource is blocked or reserved, on a per day basis.

Using **one cell per available day** would likely make computations and checking faster than keeping a list of ranges.

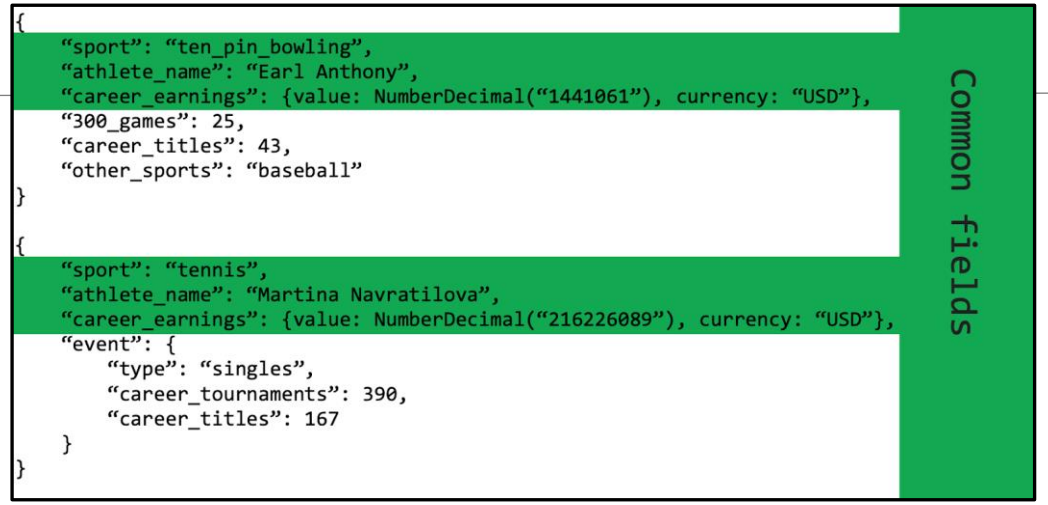
- Useful when
  - your document structure and your application simply needs to fill in data into pre-defined slots
- Pros
  - design simplification when the document structure is known in advance
- Cons
  - simplicity versus performance (size on disk)
- Examples
  - **2-dimensional structures, reservation systems**

```
{
    _id: <ObjectId>,
    month: “April”,
    year: “2019”,
    work_days:
        [
          1, 2, 3, 4, 5,
          8, 9, 10, 11, 12,
          15, 16, 17, 18, 19,
          22, 23, 24, 25, 26,
          29, 30
        ]
}
```

```
{
    _id: <ObjectId>,
    month: “April”,
    year: “2019”,
    work_days:
        [
          (1, 5),
          (8, 12),
          (15, 19),
          (22, 26),
          (29, 30)
        ]
}
```

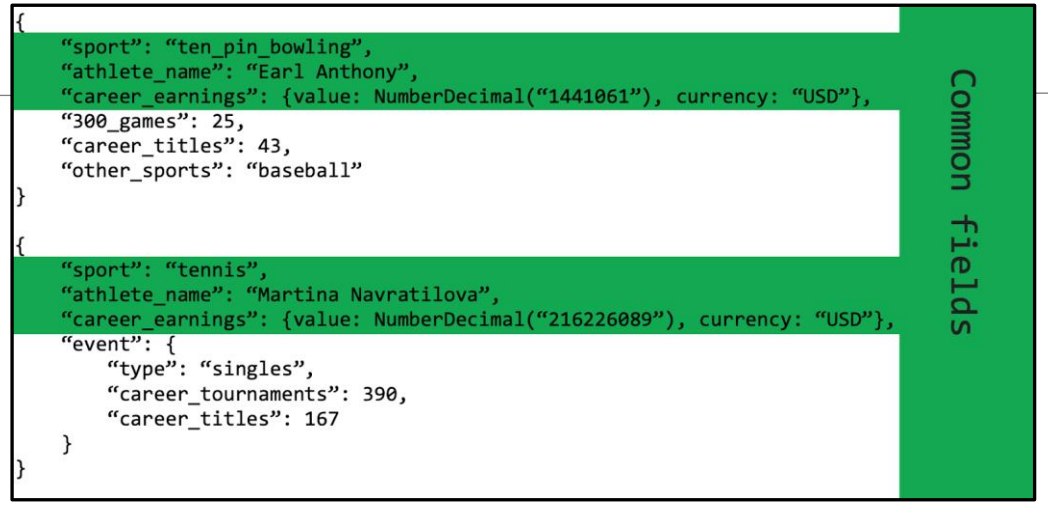
9) Polymorphic

- When all documents in a collection are of similar, but not identical, structure.
- Useful when we want to access (query) information from a single collection.
- Grouping documents together based on the queries we need to run, instead of separating the objects across tables or collections, helps improve performance.
- Example: track professional athletes across different sports.
  - If we were not using the Polymorphic Pattern, we might have a collection for Bowling Athletes and a collection for Tennis Athletes.
  - When we wanted to query on all athletes, we would need to do a time-consuming and potentially complex join.



9) Polymorphic

- Useful when
  - there are a variety of documents that have more similarities than differences
  - the documents need to be kept in a single collection
- Pros
  - Easy to implement
  - Queries can run across a single collection
- Cons
  - different code paths required in the application, based on the information in each document



- Examples
  - **Single View application**
  - **cross-company or cross-unit use cases**
  - **Wide product catalogs**
- Single View application
  - aggregates data from multiple sources into a central repository allowing customer service, insurance agents, billing, and other departments to get a 360° picture of a customer

source: https://developer.mongodb.com/how-to/polymorphic-pattern

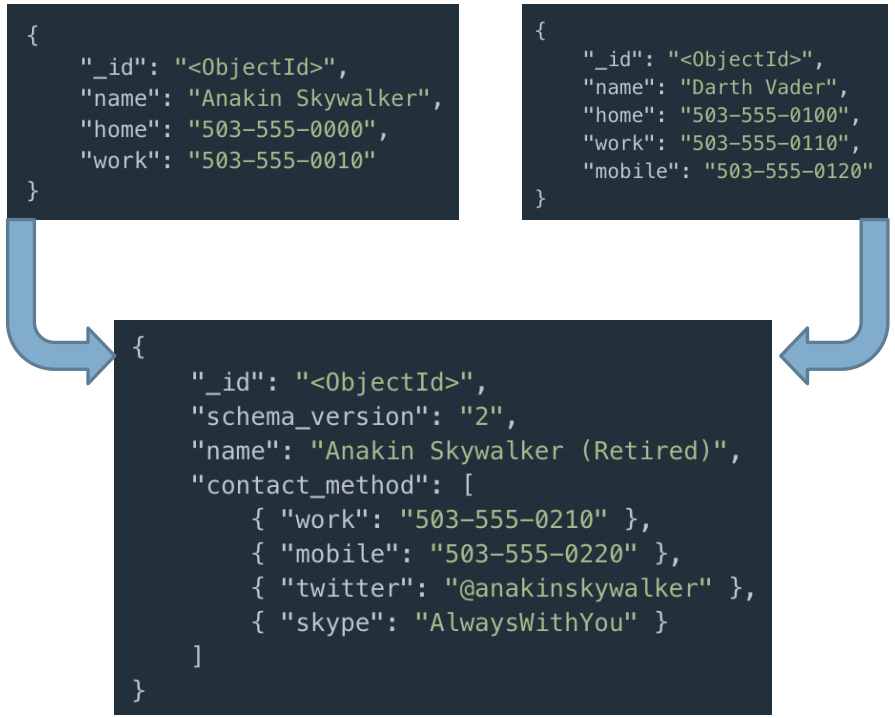
10) Schema Versioning

- Regardless of the reason behind the change, after a while, we inevitably need to **make changes** to the underlying **schema design** in our application
- This often poses challenges and perhaps some headaches in a **relational database** system
  - Typically, the application needs to be **stopped**, the database **migrated** to support the new schema and then restarted. This **downtime** can lead to poor customer experience. Additionally, what happens if the migration wasn't a complete success? **Reverting** back to the prior state is often an even larger challenge.
- In NoSQL we can use the Schema Versioning pattern to make the changes easily manageable

- Create and save the new schema to the database with a *schema_version* field. To allow our application to know how to handle this particular document.
- Avoid exploiting implicit presence of some fields.
- Increment schema_version value at each change.

10) Schema Versioning

- Useful when
  - changes to the data schema frequently occur in an application's lifetime
  - previous and current versions of documents should exist side by side in a collection
- Pros
  - no downtime needed
  - control of schema migration
  - reduced future technical debt
- Cons
  - might need two indexes for the same field during migration
- Examples
  - **customer profile**

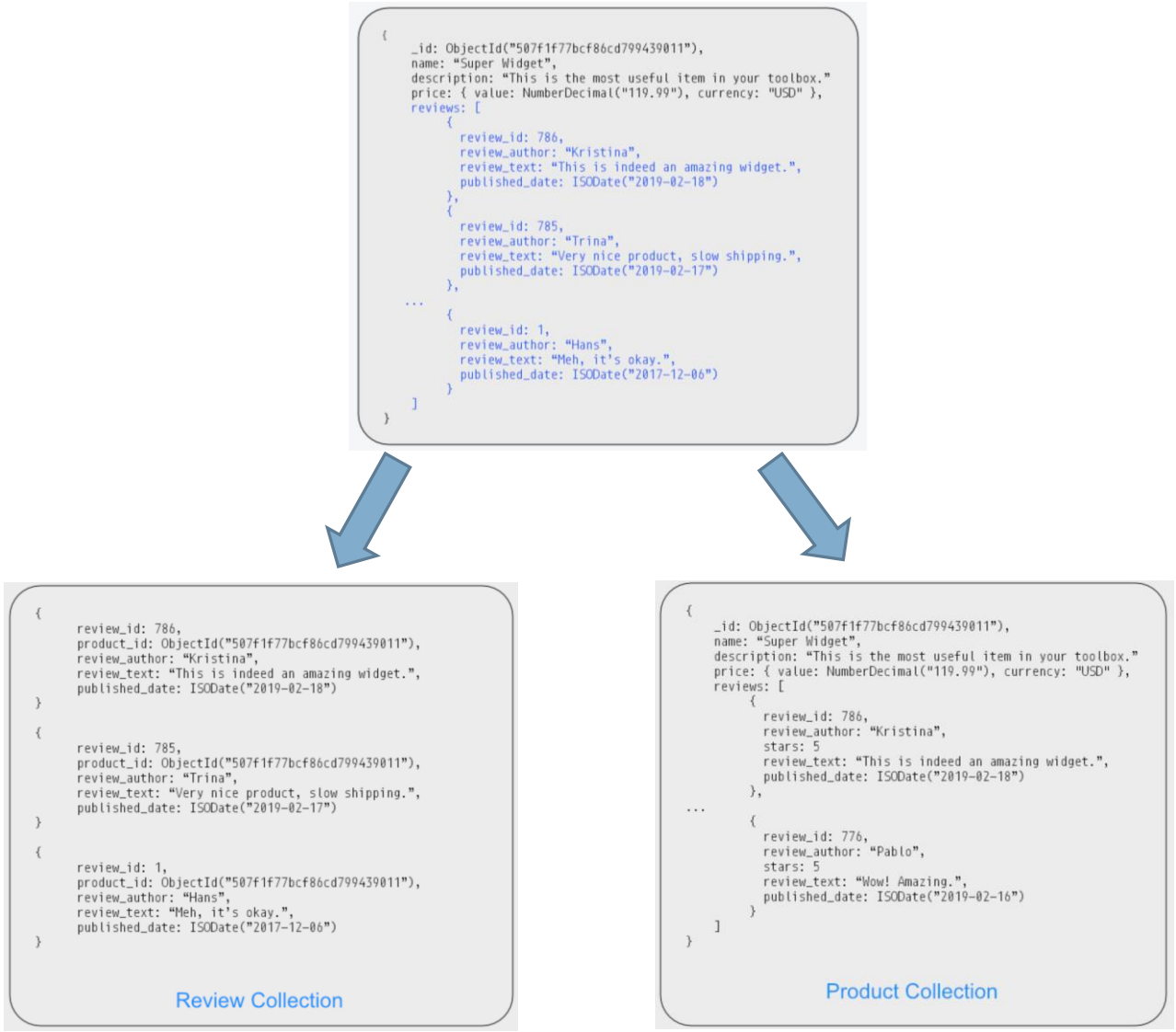


- Depending on the application and use case
  - updating all documents to the new design
  - updating when a record is accessed

source: https://www.mongodb.com/blog/post/building-with-patterns-the-schema-versioning-pattern

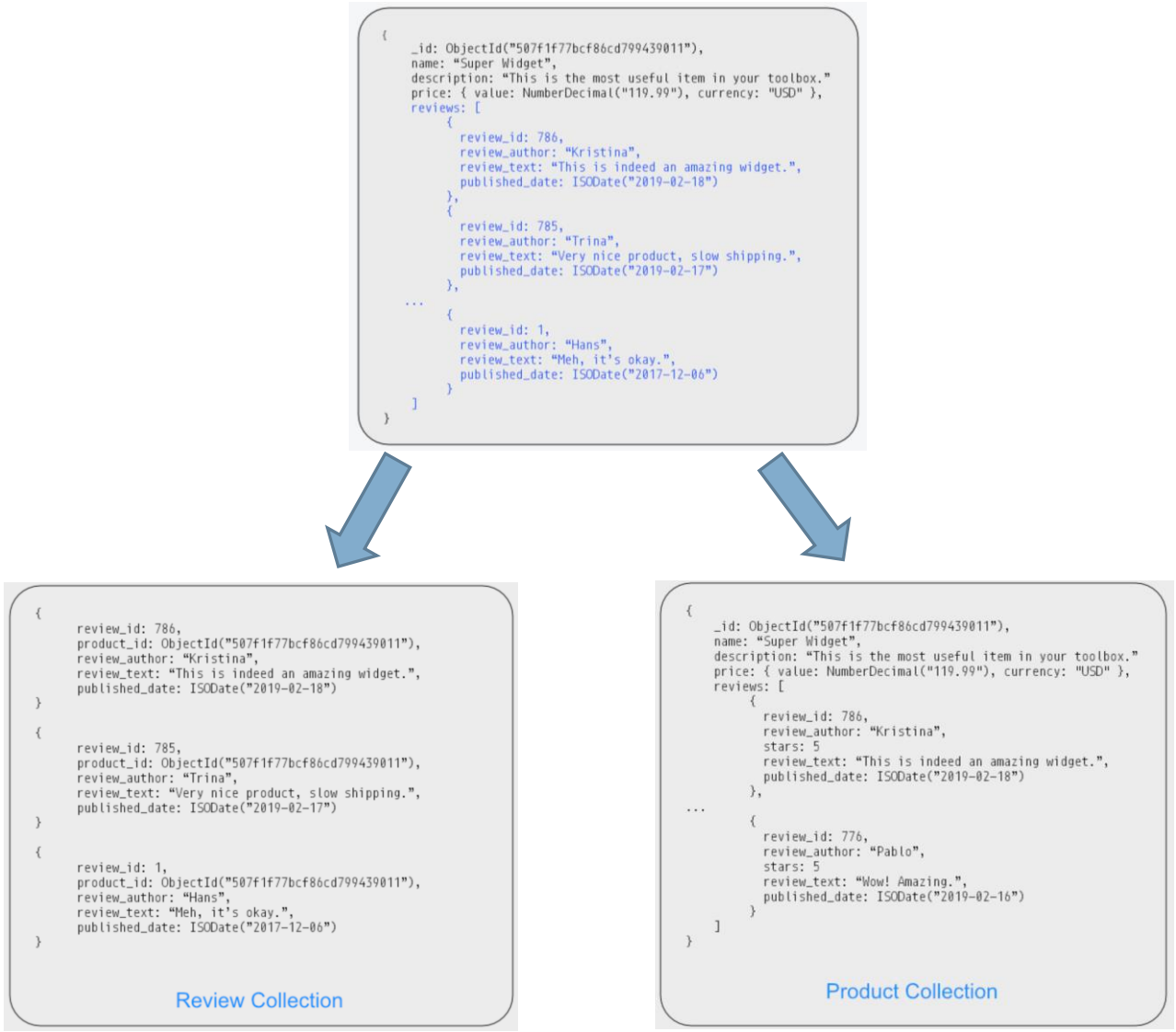
11) Subset

- When the **working set** of data and indexes grows beyond the physical RAM allotted, performance is reduced as disk accesses starts to occur and data rolls out of RAM
  - add more **RAM** to the server
  - **sharding** our collection, but that comes with additional costs and complexities
  - reduce the size of our working set with the **Subset** pattern
- Caused by **large documents** which have a lot of data that isn't actually used by the application
  - e-commerce site that has a list of **reviews** for a product.
  - accessing that product's data, we'd only need the most recent ten or so reviews.
  - pulling in the entirety of the product data with all the reviews could easily cause the working set to uselessly expand



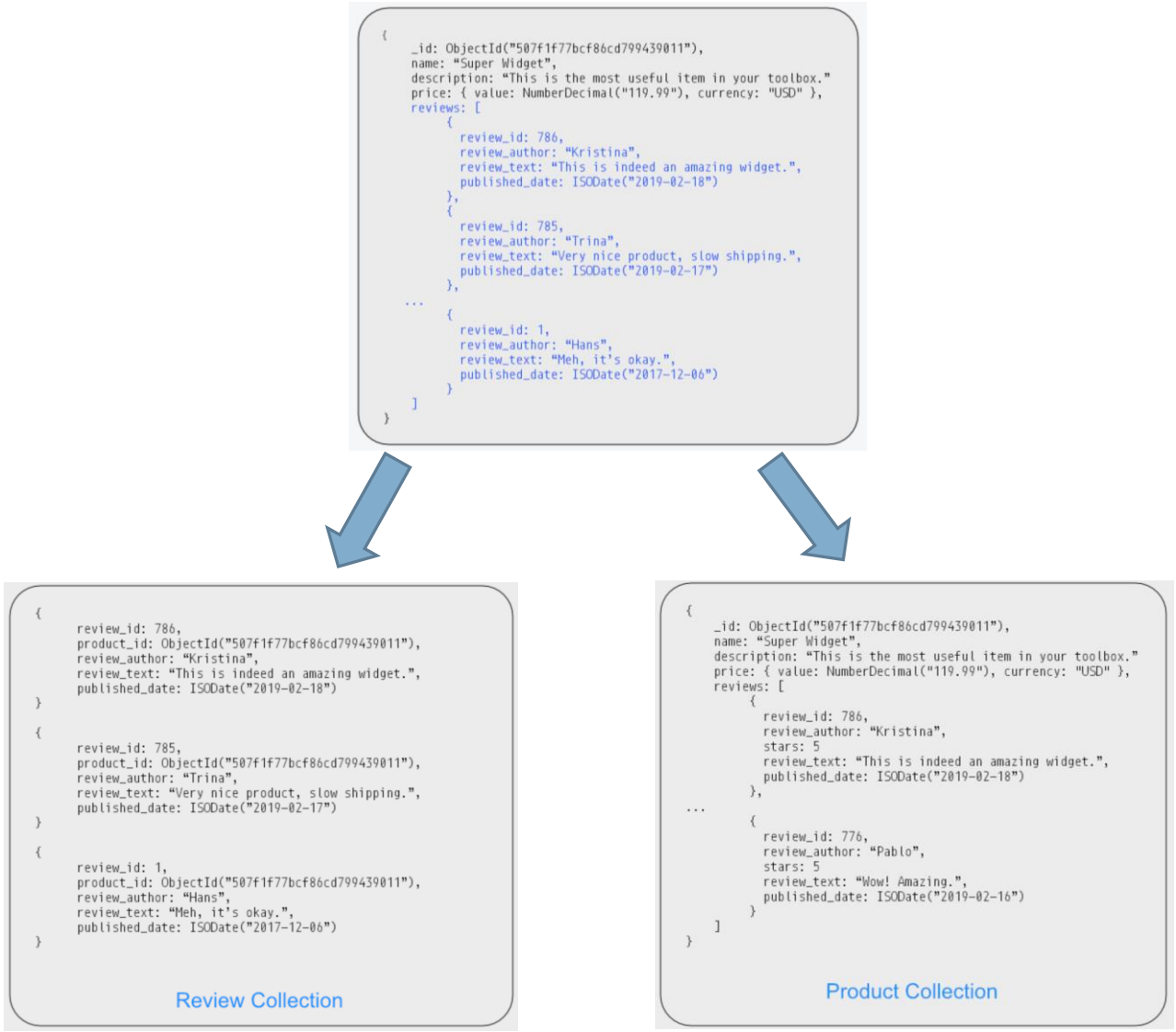
11) Subset

- Split the collection into **two collections**.
  - One collection would have the most frequently used data, e.g., current reviews
  - The other collection would have less frequently used data, e.g., old reviews, product history, etc.
- In the **Product** collection, we'll only keep the ten most recent reviews. This allows the working set to be reduced by only bringing in a portion, or subset, of the overall data.
- The additional information, reviews in this example, are stored in a separate **Reviews** collection that can be accessed if the user wants to see additional reviews.



11) Subset

- Useful when
  - the working set exceed the capacity of RAM due to large documents that have much of the data in the document not being used by the application
- Pros
  - reduction in the overall size of the working set.
  - shorter disk access time for the most frequently used data
- Cons
  - we must manage the subset
  - pulling in additional data requires additional trips to the database
- Examples
  - **reviews for a product**



12) Tree

- You would like to identify the reporting chain from an employee to the CEO
- There are many ways to represent a tree in a legacy tabular database.
  - for a node in the graph to list its parent and for a node to list its children
  - require multiple access to build the chain of nodes
- Store the full path from a node to the top of the hierarchy, as a list of the parents
  - data duplication
  - a small cost compared to the benefits you can gain from not calculating the trees all the time.
- Example: products belong to sub-categories, which are part of other super-categories.

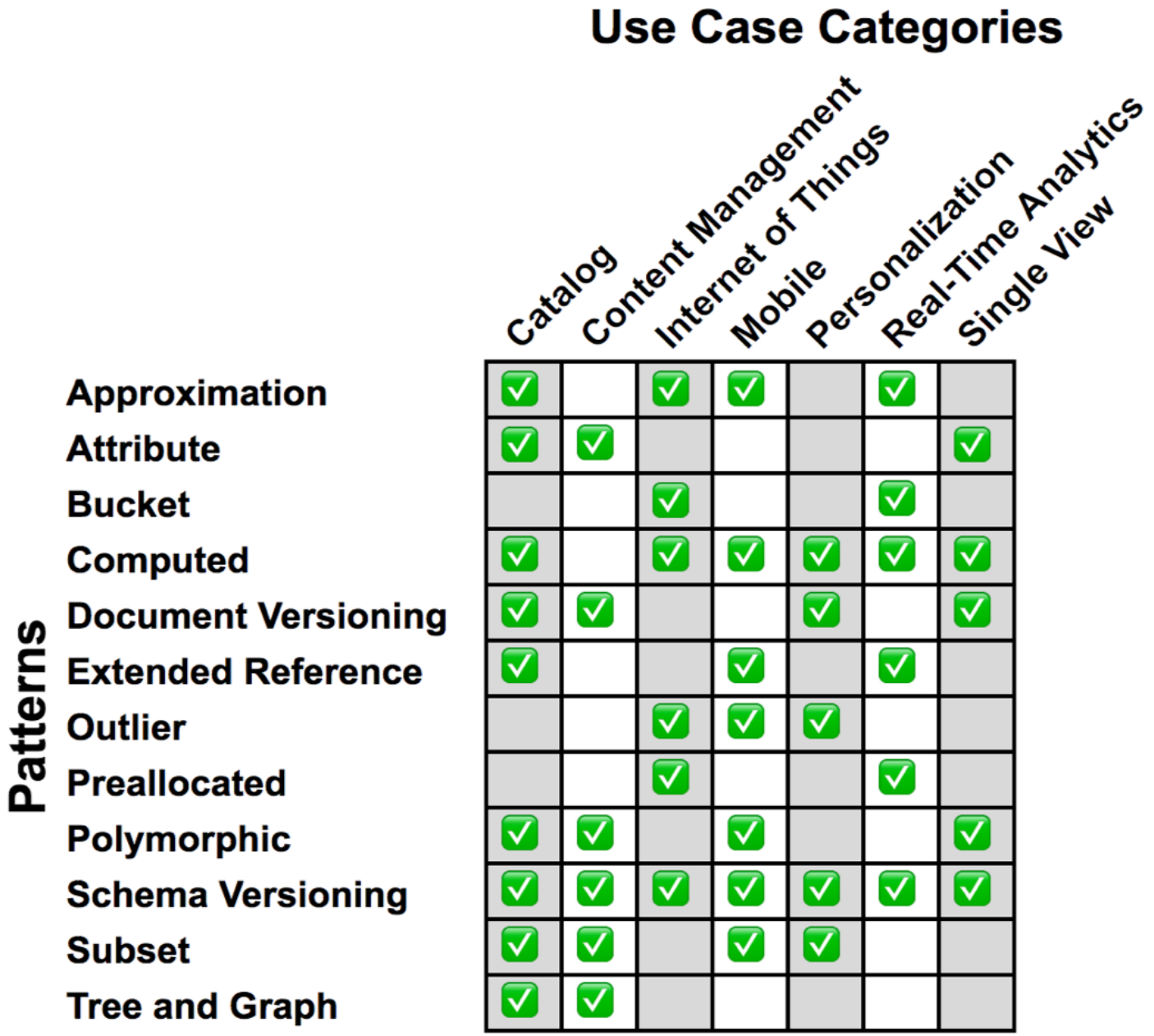
```
{
  employee_id: 5,
  name: "Jim Halpert",
  reports_to: [
     "Michael Scott",
     "Jan Levinson",
     "David Wallace"
  ]
}
```

```
{
  _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"
  ]
}
```

12) Tree

- Useful when
  - hierarchical data structure is frequently queried
- Pros
  - increased performance by avoiding multiple JOIN operations
- Cons
  - updates to the graph need to be managed in the application
- Examples
  - **product catalogs**

```
{
  _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"
  ]
}
```



Summary

- depend on the type of application
- look at the ones that are frequently used in your use case
- data schema is very dependent on your data access patterns

source: https://www.mongodb.com/blog/post/building-with-patterns-a-summary

Acknowledgment

Bibliography

For further information on the content of these slides, please refer to the book

"Design with MongoDB"

Best Models for Applications

by Alessandro Fiori

https://flowygo.com/en/projects/design-with-mongodb/

Politecnico di Torino

DBMG

Design pattern exercises (1)

DANIELE APILETTI

POLITECNICO DI TORINO

Exercise 1 – «Books and publishers»

- We want to design a MongoDB database to store the following data about 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 possibile 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
  - Indicate for each collection in the database the document structure and the strategies used for modelling.

Exercise 1 – «Books and publishers»

publisher

```
{_id: <ObjectId>,
 name:<string>,
 year:<date>,
 country: <string>,
 contacts: {email: <string>,
            tel: <string>,
            website: <string> }
}
```

book

```
{_id: <string>, // ISBN
 title: <str>,
 subtitle: <string>,
 n: <int>,
 language: <string>,
 pub_date: <date>
 publisher: {_id: <ObjectId>,
             name:<string>}
}
```

Polymorphic pattern to track only the contact methods that are available

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.

Exercise 2 – «Blog website»

- We want to design a MongoDB database for the management of a blog.
- 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.
  - Indicate for each collection in the database the document structure and the strategies used for modelling.

Exercise 2 – «Blog website»

```
user
{_id: <string>, // username
 contacts:[
          {k: <string>, // contact method
           v: <string>, // contact value
           s: <string>, // status
          ],
 nPosts: <int>,
 nComments: <int>
}
post
{_id: <ObjectId>,
 title: <str>,
 description: <string>,
 author: <string>, // username
 comments: [       // most recent comments
          {_id: <ObjectId>
           user: <string>,
           text: <string>,
           timestamp: <datetime>}
          ]
}
comment
{_id: <ObjectId>
 user: <string>,
 text: <string>,
 timestamp: <datetime>
}
```

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.

Subset pattern to track only the most recent comments.

Exercise 3 – «Breadcrumbs»

- We want to design the database for the management of a sitemap.
- 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.
  - Indicate for each collection in the database the document structure and the strategies used for modelling.

Exercise 3 – «Breadcrumbs»

page

```
{_id: < ObjectId >,
 path: <string>,
 title: <string>,
 body: <string>,
 creation: <datetime>,
 update: <datetime>,
 tags: [ <string> ],
 parent: {_id: <ObjectId>,
         path: <string>}
 ancestors: [
         {_id: <ObjectId>, // parent
          path: <string>},
         {_id: <ObjectId>, // grandpa
          path: <string>},
         …
         ]
}
```

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

Acknowledgment

Bibliography

For further information on the content of these slides, please refer to the book

"Design with MongoDB"

Best Models for Applications

by Alessandro Fiori

https://flowygo.com/en/projects/design-with-mongodb/

Politecnico di Torino

DBMG

Design pattern exercises (2)

DANIELE APILETTI

POLITECNICO DI TORINO

Exercise 4 – Parcel delivery (1)

- 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.

Exercise 4 – Parcel delivery (2)

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

Exercise 4 – Parcel delivery

```
Parcel
{_id: <string>, // barcode
 dimensions: { // also 1st-level attribute is fine
     width: <number>,
     height: <number>,
     depth: <number>,
     weight: <number>
 },
 recipient: {
     _id: <string>, // e.g., SSN, id of the customer
     street: <string>,
     number: <string>,
     zip_code: <string>,
     city: <string>,
     province: <string>
 },
 sender:{
     _id: <string>,// e.g., SSN, _id of the customer
     street: <string>,
     number: <string>,
     zip_code: <string>,
     city: <string>,
     province: <string>
 },
 father_pos: <string>, // code of area/lane/rack/shelf
 locations: [
     <string> // 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 tree-pattern ancestors is required. The parent ancestor of the tree-pattern is optional.

(No collection for the areas, since no data are tracked except their code)

Exercise 4 – Parcel delivery

```
Customers
{_id: <string>, // fiscal code
 name: <string>,
 surname: <string>,
 email: <string>,
 tel: <string>,
 addresses:{
     'home': {
          street_name: <string>,
          street_num: <string>,
          postal_code: <int>,
          city: <string>,
          province: <string>,
          country: <string>
     },
     'billing':{
          street_name: <string>,
          street_num: <string>,
          postal_code: <int>,
          city: <string>,
          province: <string>,
          country: <string>
     },
     'work': {...}
 }
}
```

Attribute pattern (optional) for the addresses attribute.

Exercise 5 – «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.

Exercise 5 – «Museum exhibitions»

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

Exercise 5 – «Museum exhibitions»

```
Museums
{_id: ObjectId(),
 name: <string>,
 address: {
     street: <string>,
     number: <string|number>,
     postal_code: <number>,
     city: <string>,
     province: <string>,
     geo_ref: {
          type: <string>,
          coordinates: [ <number> ]
     }
 },
 phone: <string>,
 website: <string> // optional
}
Items
{ _id: <number>,
 title: <string>,
 description: <string>,
 authors: [ <string> ],
 category: <string>,
 features: [ {k: <string>, v: <string>, u: <string>} ],
 museum: {
     _id: ObjectId(),
     name: <string>
 }
}
```

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.

Exercise 5 – «Museum exhibitions»

```
Exhibitions
{_id: ObjectId(),
 title: <string>,
 description:<string>,
 curators: [ <string> ],
 events: [
    {start: <date>,
     end: <date>,
     museum:{
         _id: itemId(),
         name: <string>,
         geo_ref: {
             type: <string>,
             coordinates: [ <number> ]
         }
    }
 ],
 items: [<number> ] // _id of items }
}
```

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

Acknowledgment

Bibliography

For further information on the content of these slides, please refer to the book

"Design with MongoDB"

Best Models for Applications

by Alessandro Fiori

https://flowygo.com/en/projects/design-with-mongodb/

EXERCISES

EXERCISE 1 – BOOKS

We are required to design a MongoDB database to store the following data about books.

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.).

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

EXERCISE 1 – BOOKS

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

EXERCISE 2 – 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.

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

EXERCISE 2 – PROPERTY RENTAL REVIEWS

Properties

```
{_id: <ObjectId>,
 name: <string>,
 price: {  amount: <double>,
           currency: <string>},
 location:{
           postal_code: <int>,
           city: <string>,
           country: <string>,
          },
 reviews:[   // top 10 most recent
           {review_id: <ObjectId>
            text: <string>,
            timestamp: <datetime>,
            grade: <double>}
         ],
 nReviews: <int>,
 avgGrade: <double>
}
```

Reviews

```
{_id: <ObjectId>,
 property_id: <ObjectId>,
 text: <string>,
 timestamp: <datetime>,
 grade: <double>,
 author: { name: <string>,
           email: <string>}
}
```

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.

EXERCISE 3 – INSURANCE

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 3 – INSURANCE

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

ACKNOWLEDGMENT

BIBLIOGRAPHY

For further information on the content of these slides, please refer to the book

“Design with MongoDB”

Best Models for Applications

by Alessandro Fiori

https://flowygo.com/en/projects/design-with-mongodb/

EXERCISES

EXERCISE 1 – PROPERTY LISTING

A) Select all properties that have wifi and dryer as amenities sorted by increasing price. Display only the name, url and price.

SOLUTION

```
db.listingsAndReviews.find(
     { amenities:
          {$all: ["Wifi", "Dryer"] }
     },
{_id:0, name: 1, price:1, url: 1}
).sort( {price:1} )
```


```
{
  "_id": "10006546",
  "url": "https://www.airbnb.com/rooms/10006546",
  "name": "Ribeira Charming Duplex",
  "description": "Fantastic duplex apartment with . . .",
  "property_type": "House",
  "minimum_nights": 2,
  "maximum_nights": 30,
  "beds": 5,
  "number_of_reviews": 51,
  "amenities": [
    "TV",
    "Wifi",
    "Smoking allowed",
    "Pets allowed",
    "Dryer",
    "Heating"
  ],
  "price": 80,
  "host": {
    "host_id": "51399391",
    "name": "Ana&Gonçalo",
    "city": "Porto",
    "country": "Portugal"
  }
}
```

EXERCISE 1 – PROPERTY LISTING

B) For all House type properties located in the city of Turin, set the minimum number of nights to 3. If the minimum number of nights is greater than 3, it should not be updated.

SOLUTION

```
db.listingsAndReviews.updateMany(
 {"property_type": "House",
  "host.city": "Turin",
  "minimum_nights": {lt: 3 }
 },
 {$set: {"minimum_nights": 3}}
)
```


```
{
  "_id": "10006546",
  "url": "https://www.airbnb.com/rooms/10006546",
  "name": "Ribeira Charming Duplex",
  "description": "Fantastic duplex apartment with . . .",
  "property_type": "House",
  "minimum_nights": 2,
  "maximum_nights": 30,
  "beds": 5,
  "number_of_reviews": 51,
  "amenities": [
    "TV",
    "Wifi",
    "Smoking allowed",
    "Pets allowed",
    "Dryer",
    "Heating"
  ],
  "price": 80,
  "host": {
    "host_id": "51399391",
    "name": "Ana&Gonçalo",
    "city": "Porto",
    "country": "Portugal"
  }
}
```

EXERCISE 1 – PROPERTY LISTING

C) Calculate the median price of all House type properties.

SOLUTION

```
db.listingsAndReviews.aggregate(
      [
{     $match: {property_type: "House"}},
{     $sort: {price: 1}},
{     $group: {
            '_id': null,
            'value': {'$push': '$price'}}},
{     $project: {
           _id: 1,
           "median": {$arrayElemAt:["$value", {$floor:
{$multiply: [0.50,{$size: "$value"} ]}} ]}}}
]
)
```


```
{
  "_id": "10006546",
  "url": "https://www.airbnb.com/rooms/10006546",
  "name": "Ribeira Charming Duplex",
  "description": "Fantastic duplex apartment with . . .",
  "property_type": "House",
  "minimum_nights": 2,
  "maximum_nights": 30,
  "beds": 5,
  "number_of_reviews": 51,
  "amenities": [
    "TV",
    "Wifi",
    "Smoking allowed",
    "Pets allowed",
    "Dryer",
    "Heating"
  ],
  "price": 80,
  "host": {
    "host_id": "51399391",
    "name": "Ana&Gonçalo",
    "city": "Porto",
    "country": "Portugal"
  }
}
```

EXERCISE 2 – EXAMS

A) For each type of test, calculate the minimum, maximum, and average grades taken by students. Display this information only for the types with more than 50 tests taken.

SOLUTION

```
db.grades.aggregate(
[{$unwind: {path: '$scores'}},
{$group: {
  _id: '$scores.type',
  avg: {$avg: '$scores.score'},
  min: {$min: '$scores.score'},
  max: {$max: '$scores.score'},
  count: {$sum: 1}}},
{$match: { count: {$gt:50}}}]
)
```


```
{
  "_id": "56d5f7eb604eb380b0d8d8d2",
  "student_id": 0,
  "scores": [
    {
      "type": "exam",
      "score": 41.25131199553351
    },
    {
      "type": "quiz",
      "score": 91.7351500084582
    },
    {
      "type": "homework",
      "score": 24.198828271948415
    },
    {
      "type": "homework",
      "score": 79.77471812670814
    }
  ],
  "class_id": 391
}
```

EXERCISE 2 – EXAMS

B) Considering only the sufficient scores (grade equal to or higher than 60) of the exam type, calculate for each class the number of exams taken and the average of the grades obtained.

SOLUTION

```
db.grades.aggregate(
[{$unwind: {
  path: "$scores",
  }},
{$match: {
  "scores.type":"exam",
  "scores.score": {$gte: 60}
}},
{$group: {
  _id: "$class_id",
  c: {$sum:1},
  avg: {$avg:"$scores.score"}
}}]
```

```
{
  "_id": "56d5f7eb604eb380b0d8d8d2",
  "student_id": 0,
  "scores": [
    {
      "type": "exam",
      "score": 41.25131199553351
    },
    {
      "type": "quiz",
      "score": 91.7351500084582
    },
    {
      "type": "homework",
      "score": 24.198828271948415
    },
    {
      "type": "homework",
      "score": 79.77471812670814
    }
  ],
  "class_id": 391
}
```

EXERCISE 2 – EXAMS

C) Enter the certificate grade of 85 for the student with identifier of 1000 enrolled in grade 150. Also set a flag named "completed".

SOLUTION

```
db.grades.updateOne({ "student_id": 1000,
class_id:150},
{$push: {scores: {"type": "certificate", "grade": 85
}}, $set: {completed: true} })
```


```
{
  "_id": "56d5f7eb604eb380b0d8d8d2",
  "student_id": 0,
  "scores": [
    {
      "type": "exam",
      "score": 41.25131199553351
    },
    {
      "type": "quiz",
      "score": 91.7351500084582
    },
    {
      "type": "homework",
      "score": 24.198828271948415
    },
    {
      "type": "homework",
      "score": 79.77471812670814
    }
  ],
  "class_id": 391
}
```