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

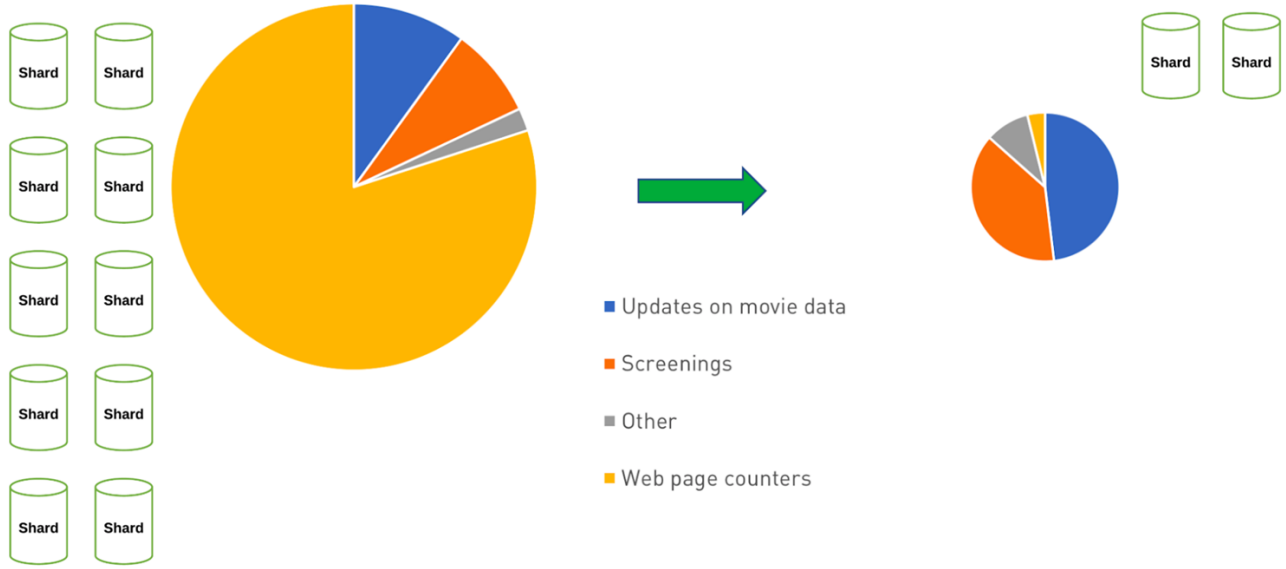
1. Approximation
2. Attribute
3. Bucket
4. Computed
5. Document Versioning
6. Extended Reference
7. Outlier
8. Pre-allocation
9. Polymorphic
10. Schema Versioning
11. Subset
12. Tree

"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.



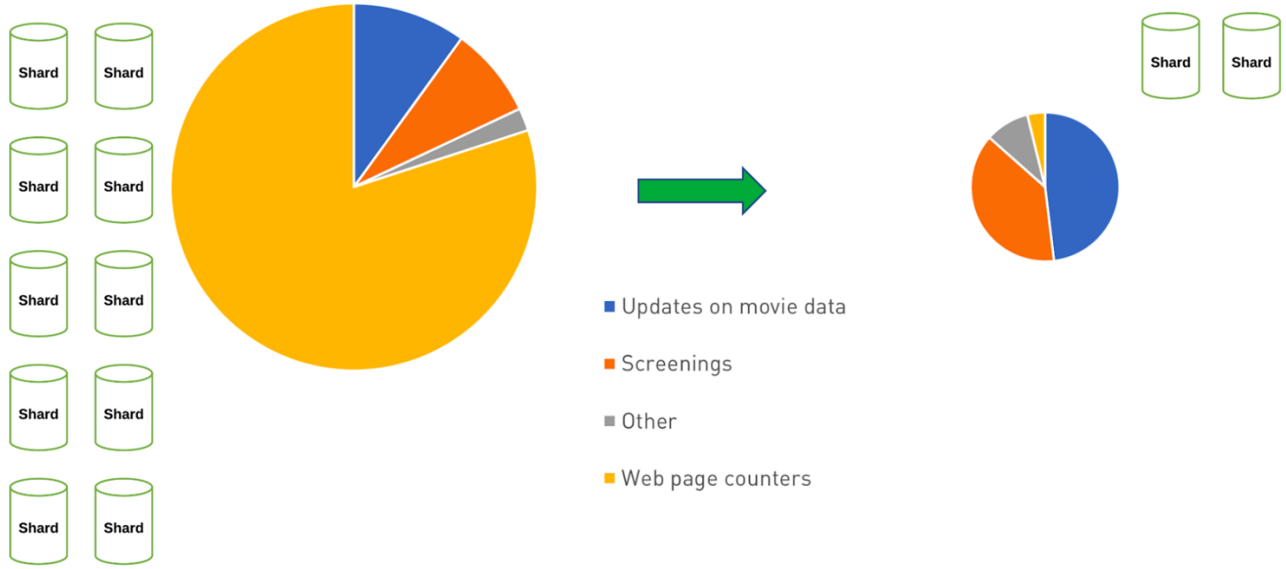
Examples

- **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
}
```

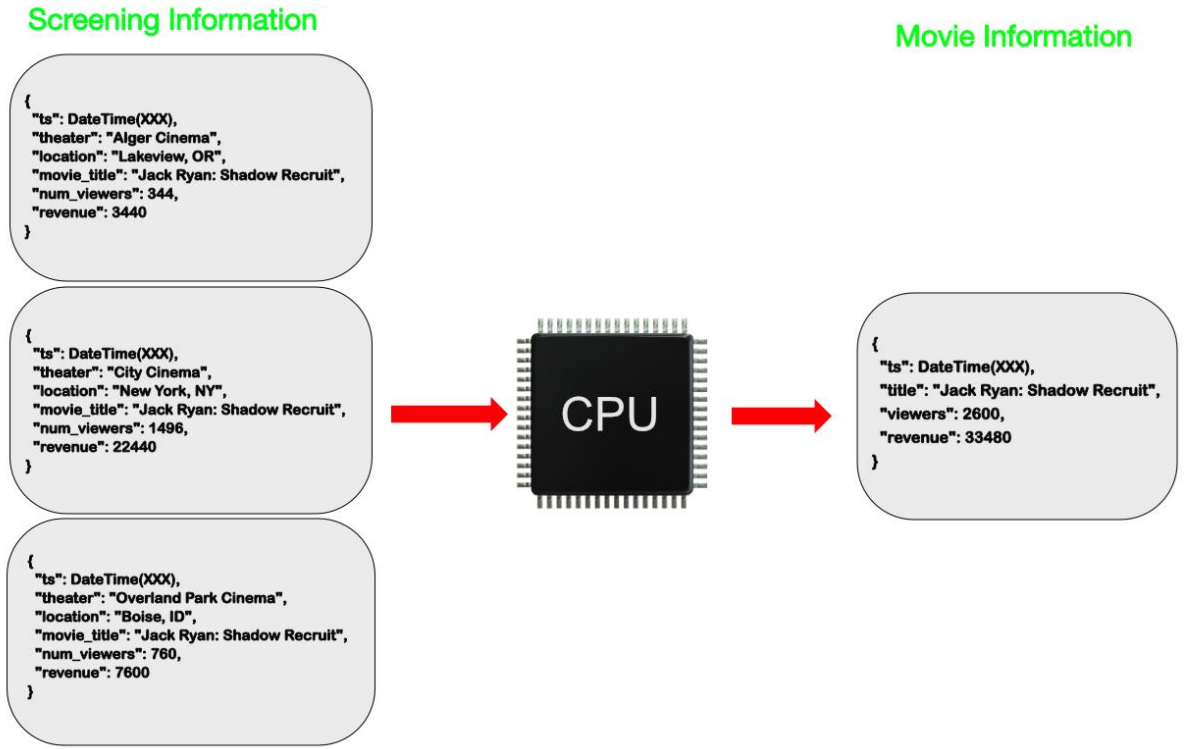
3) Bucket

- 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
}
```

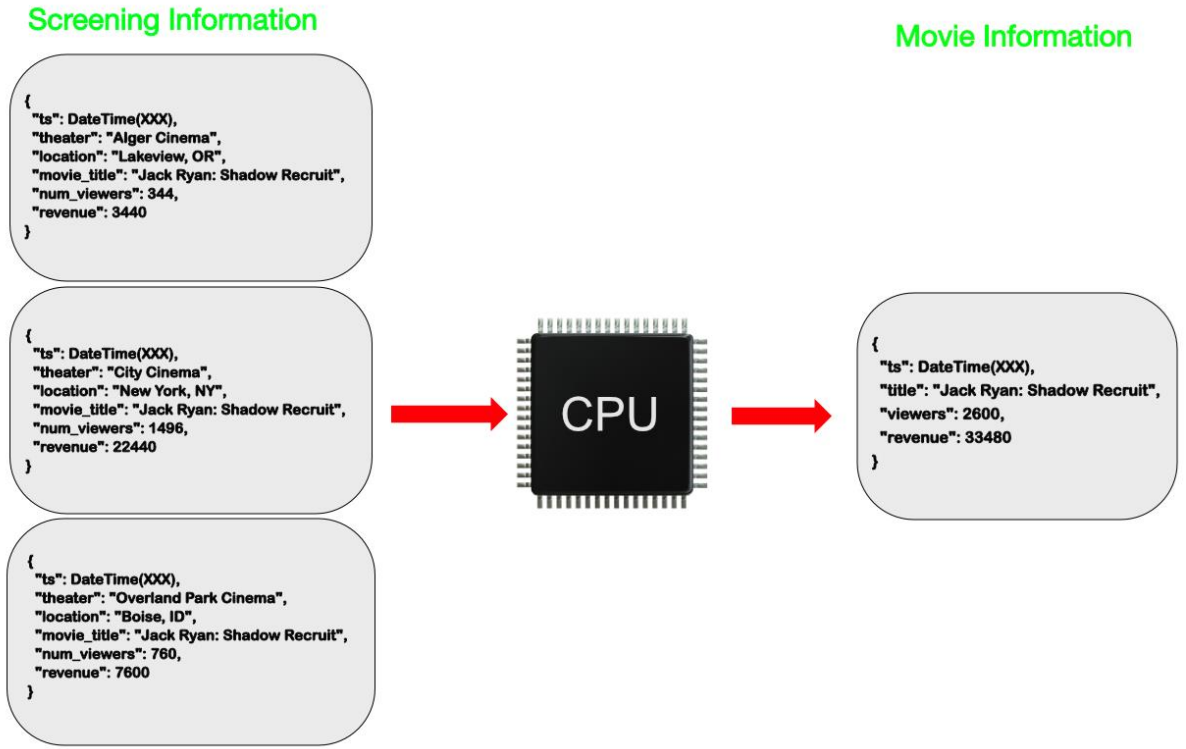
4) Computed

- 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.



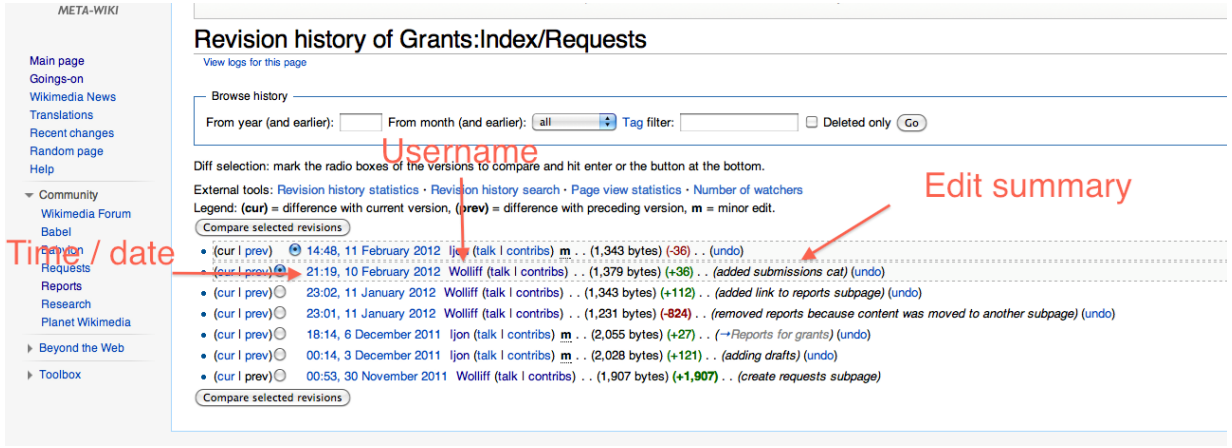
4) Computed

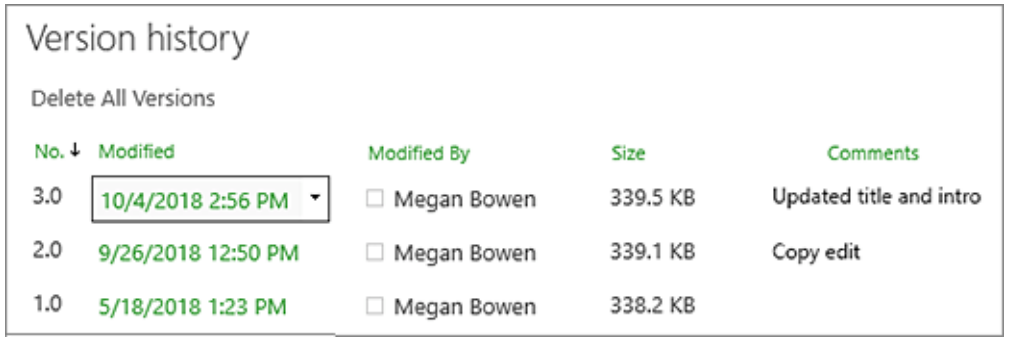
- Useful when
  - very read-intensive data access patterns
  - data needs to be repeatedly computed by the application
  - computation done in conjunction with any update or at defined intervals - every hour for example
- Pros
  - reduction in CPU workload for frequent computations
- Cons
  - it may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



5) Document Versioning

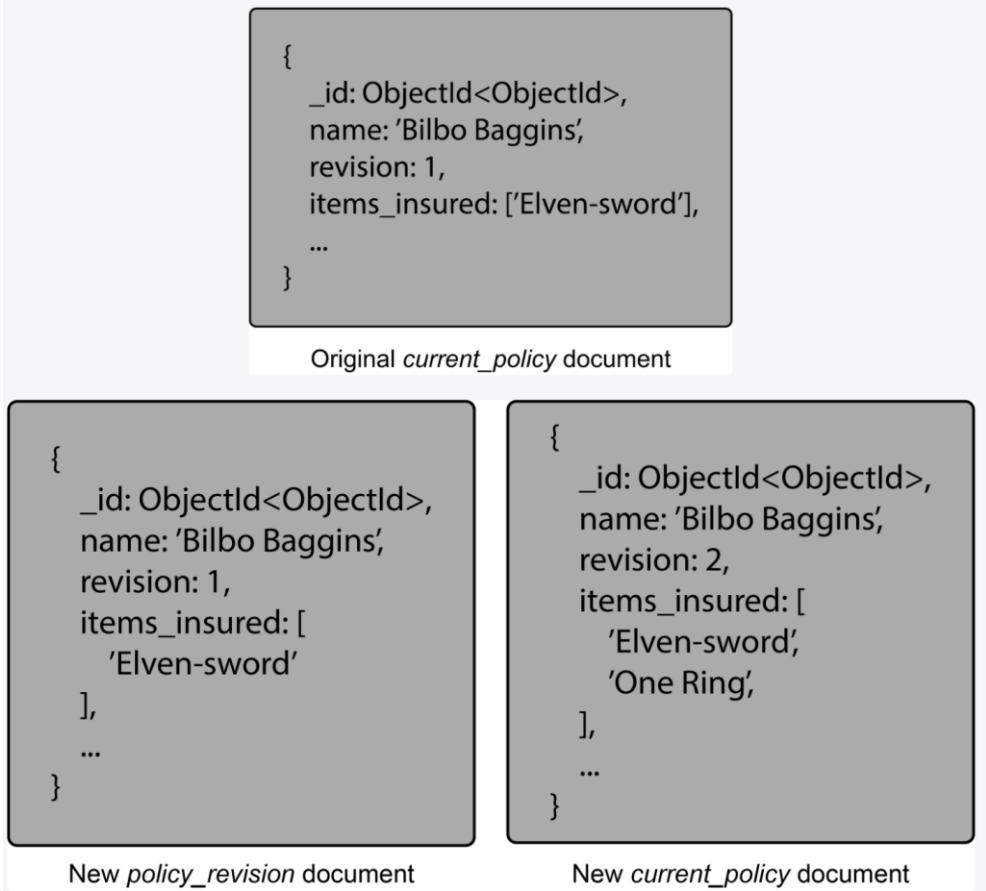
- In most cases we query **only the latest** state of the data.
  - What about situations in which we need to query previous states of the data?
  - What if we need to have some functionality of version control of our documents?
- Goal: keeping the version history of documents available and usable
- 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

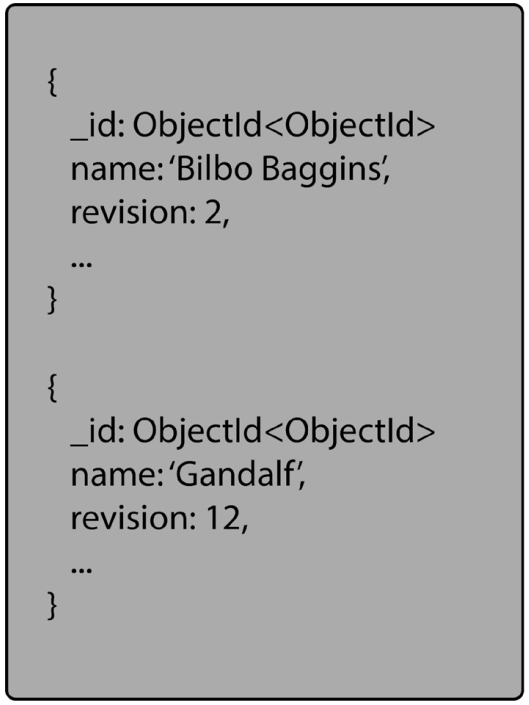
- An insurance company might make use of this pattern.
  - Each customer has a "standard" policy and a second portion that is specific to that customer.
  - This second portion would contain a **list of policy add-ons** and a list of specific items that are being insured.
- As the customer changes which 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.



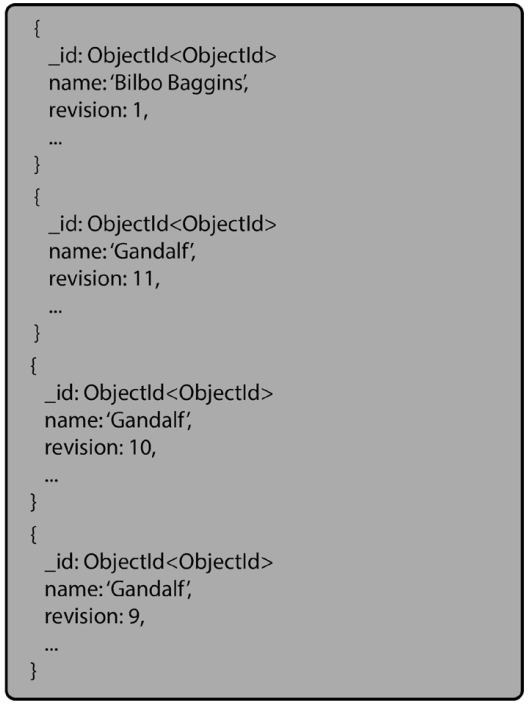
5) Document Versioning

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,
    ...
}
```

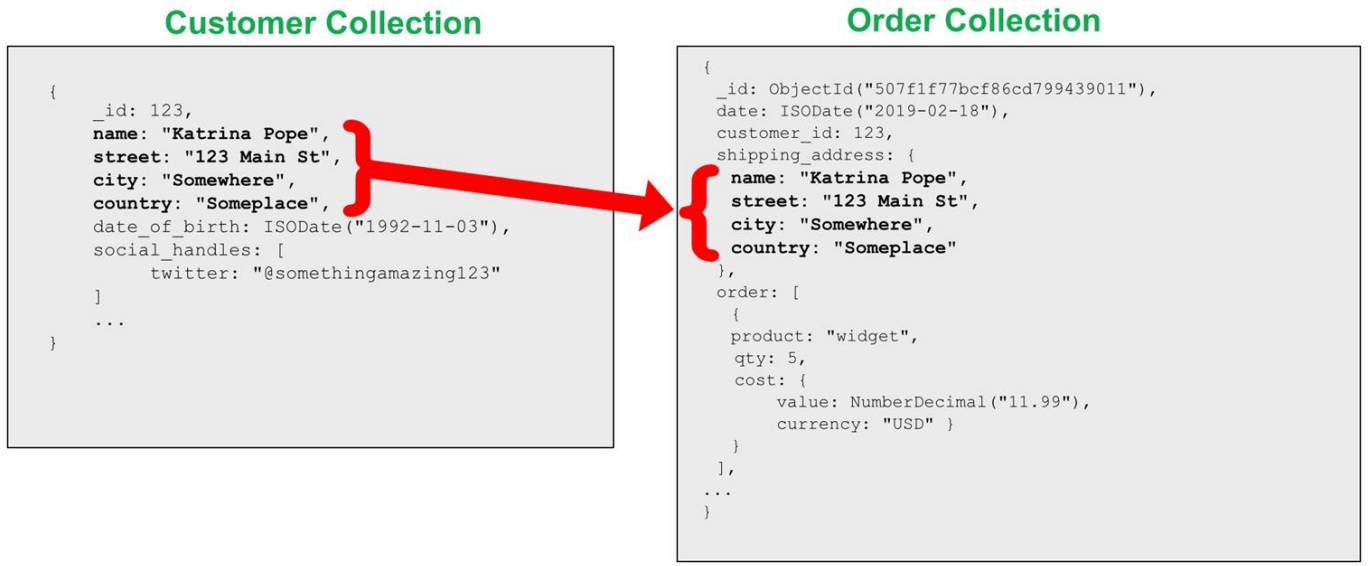
- However, the full retrieval of an order requires to **join** data from different entities
- A customer can have N orders, creating a 1-N relationship
- Embedding all the customer information inside each order
  - avoids the JOIN operation
  - results in a lot of **duplicated** information
  - not all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