MONGODB DESIGN PATTERNS 2

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

- Approximation
- Attribute
- Bucket
- Computed
- Document Versioning
- Extended Reference
- Outlier
- Pre-allocation
- Polymorphic
- Schema Versioning
- Subset
- Tree

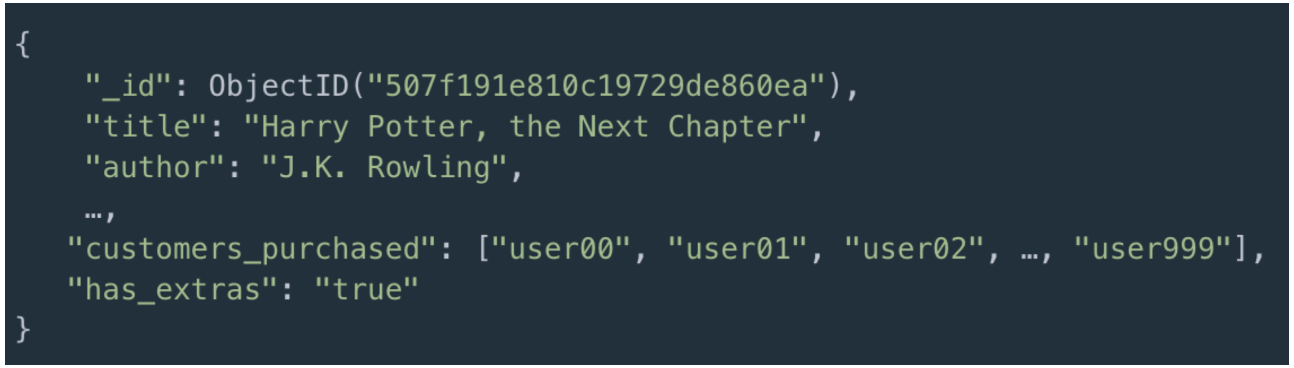
"MongoDB is **schema-less**. In fact, schema design is very important in MongoDB. The hard fact is that most performance issues we've found trace back to poor **schema design**."

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

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

OUTLIER

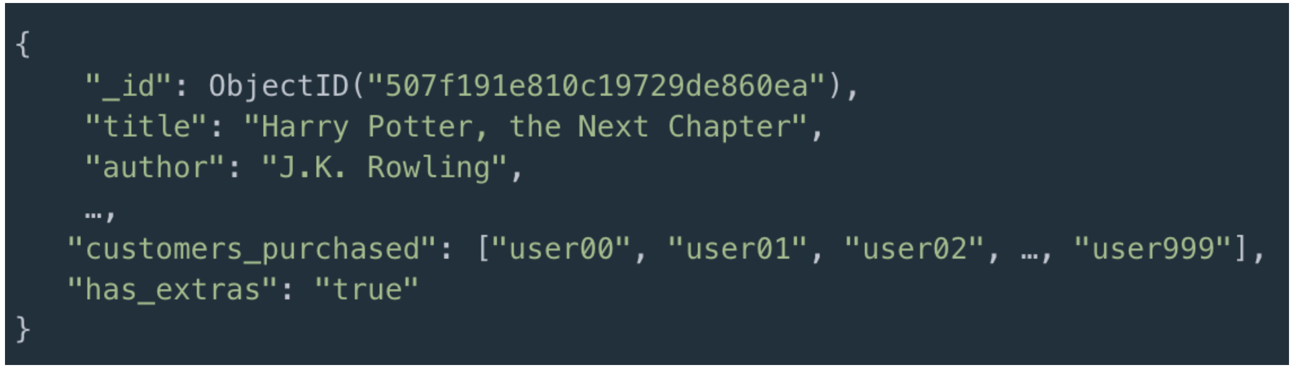
- E-Commerce selling books
  - who has purchased a particular book?
  - store an array of *user_id* who purchased the book, in each book document
- You have a solution that works for 99.99% of the cases, but what happens when a top-seller book is released?
  - You cannot store millions of user_ids due 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.

OUTLIER



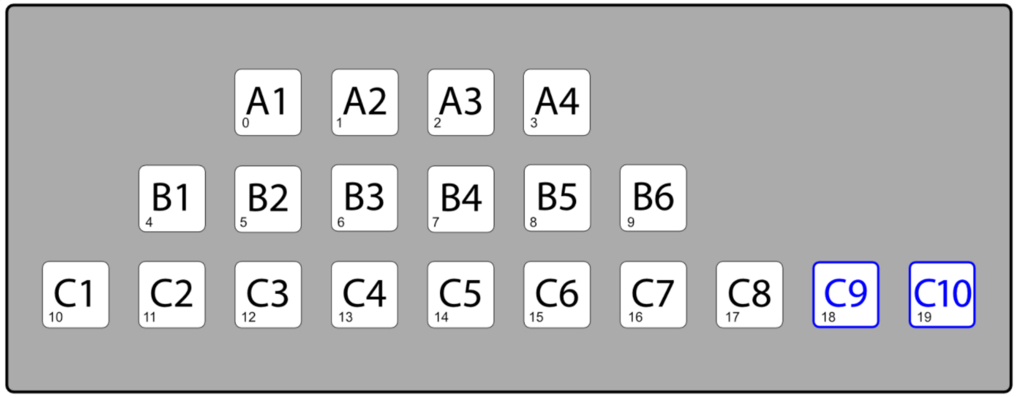
```
{
    "_id": ObjectID("507f191e810c19729de860ea"),
    "title": "Harry Potter, the Next Chapter",
    "author": "J.K. Rowling",
    …,
   "customers_purchased": ["user00", "user01", "user02", …, "user999"],
   "has_extras": "true"
}
```

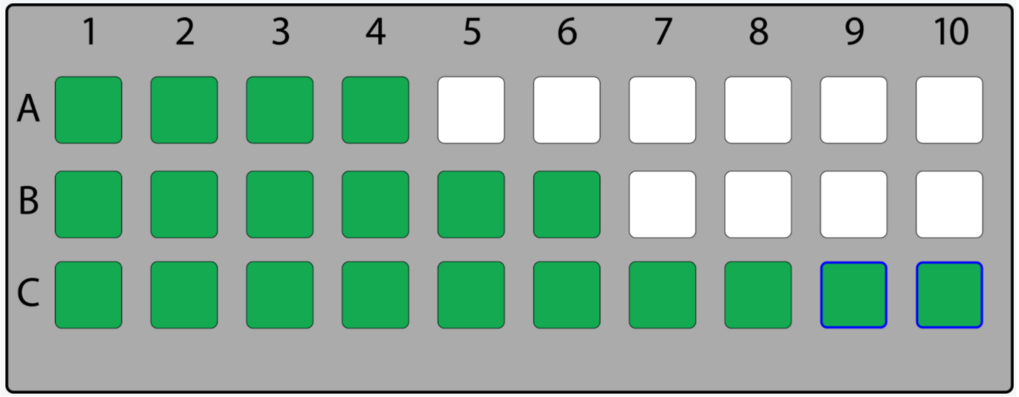
- Useful when
  - few queries or documents that don't fit into the rest of your typical data patterns
- Pros
  - prevents a few documents or queries from determining an application's solution.
  - queries are tailored for "typical" use cases, but outliers are still addressed
- Cons
  - often tailored 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

- 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.

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.



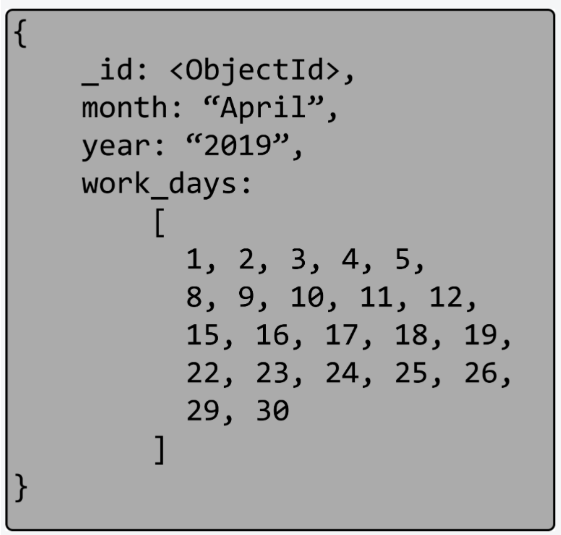


PRE-ALLOCATION

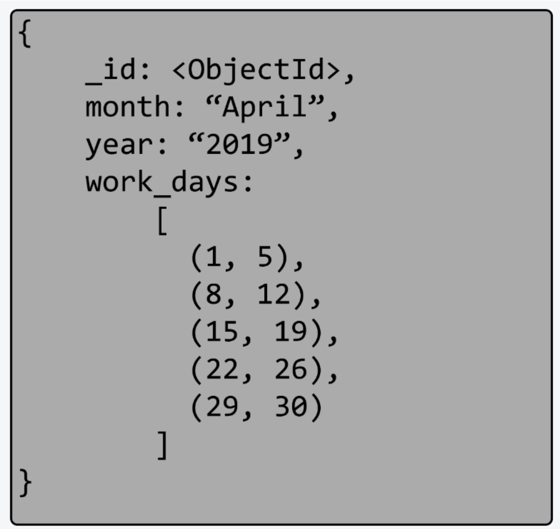
Another example: a reservation system where a resource is blocked or reserved, on a per day basis.

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)
        ]
}
```

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.

```
{
    “sport”: “ten_pin_bowling”,
    “athlete_name”: “Earl Anthony”,
    “career_earnings”: {value: NumberDecimal(“1441061”), currency: “USD”},
    “300_games”: 25,
    “career_titles”: 43,
    “other_sports”: “baseball”
}

{
    “sport”: “tennis”,
    “athlete_name”: “Martina Navratilova”,
    “career_earnings”: {value: NumberDecimal(“216226089”), currency: “USD”},
    “event”: {
        “type”: “singles”,
        “career_tournaments”: 390,
        “career_titles”: 167
    }
}
```

Common fields

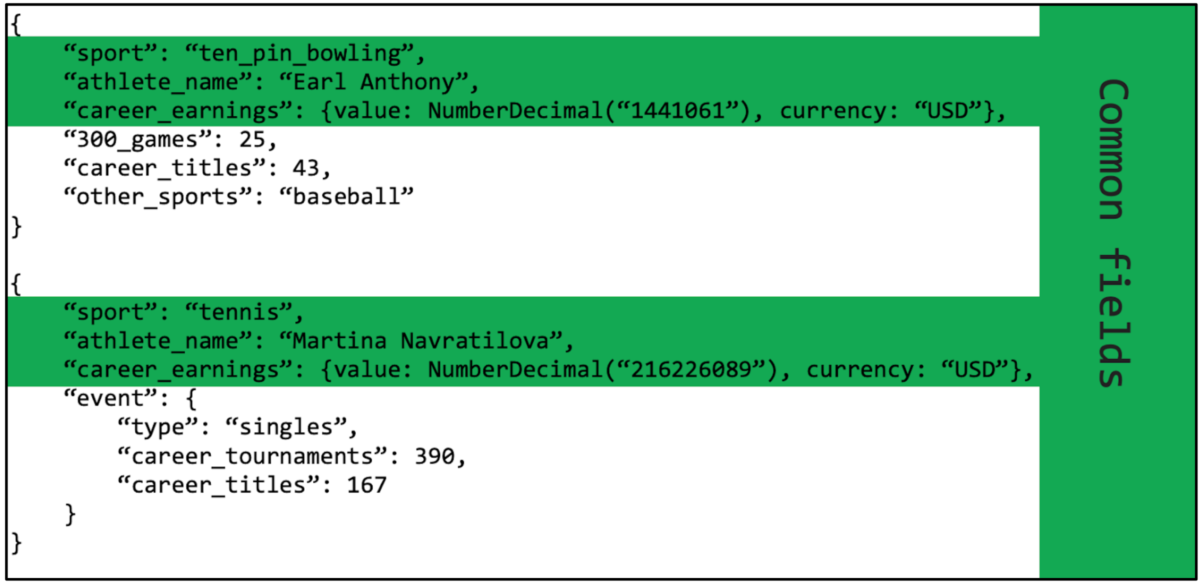
```
{
    “sport”: “tennis”,
    “athlete_name”: “Martina Navratilova”,
    “career_earnings”: {value: NumberDecimal(“216226089”), currency: “USD”},
    “career_tournaments”: 390,
    “career_titles”: 167,
    “event”: [ {
        “type”: “singles”,
        “career_tournaments”: 390,
        “career_titles”: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

...

}
```

Polymorphic Sub-Documents

POLYMORPHIC



```
{
    “sport”: “ten_pin_bowling”,
    “athlete_name”: “Earl Anthony”,
    “career_earnings”: {value: NumberDecimal(“1441061”), currency: “USD”},
    “300_games”: 25,
    “career_titles”: 43,
    “other_sports”: “baseball”
}

{
    “sport”: “tennis”,
    “athlete_name”: “Martina Navratilova”,
    “career_earnings”: {value: NumberDecimal(“216226089”), currency: “USD”},
    “event”: {
        “type”: “singles”,
        “career_tournaments”: 390,
        “career_titles”: 167
    }
}
```

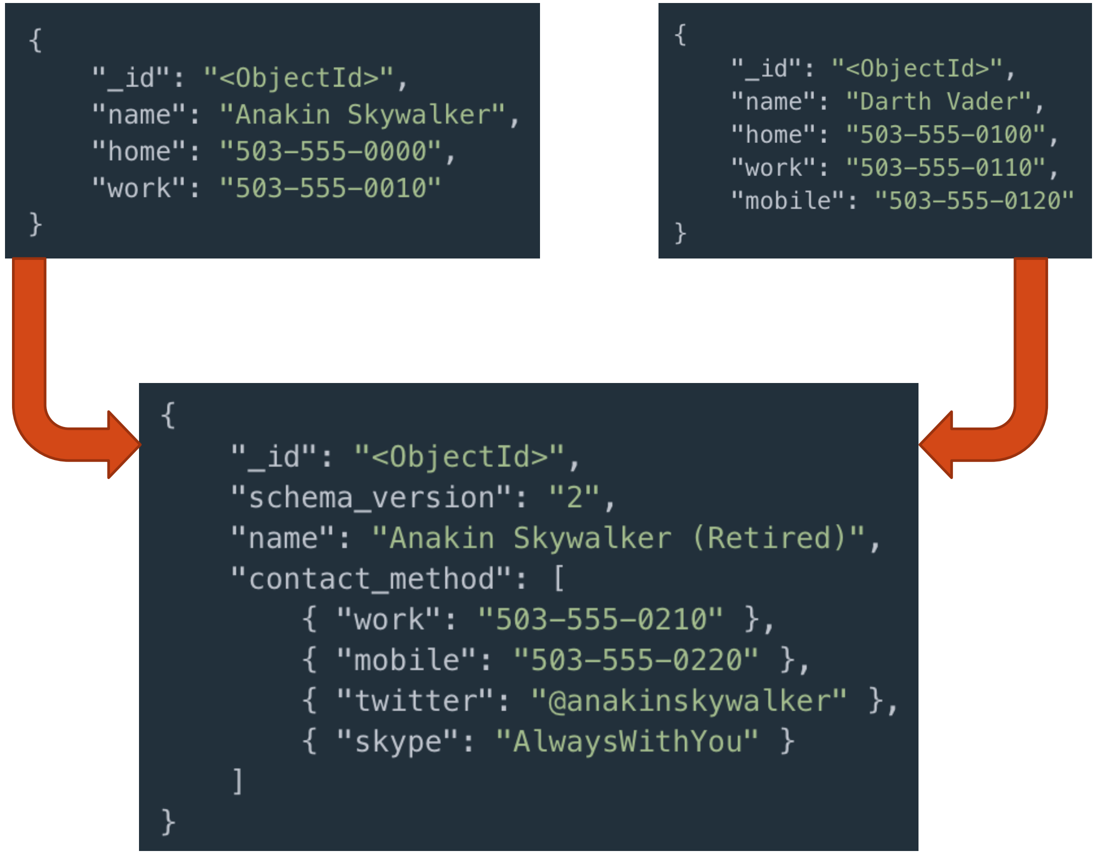
- Useful when
  - there are a variety of documents that have more similarities than differences
  - the documents need to be kept in a single collection
- Pros
  - Easy to implement
  - Queries can run across a single collection
- Cons
  - different code paths required in the application, based on the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

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



```
{
    "_id": "<ObjectId>",
    "name": "Anakin Skywalker",
    "home": "503-555-0000",
    "work": "503-555-0010"
}
```

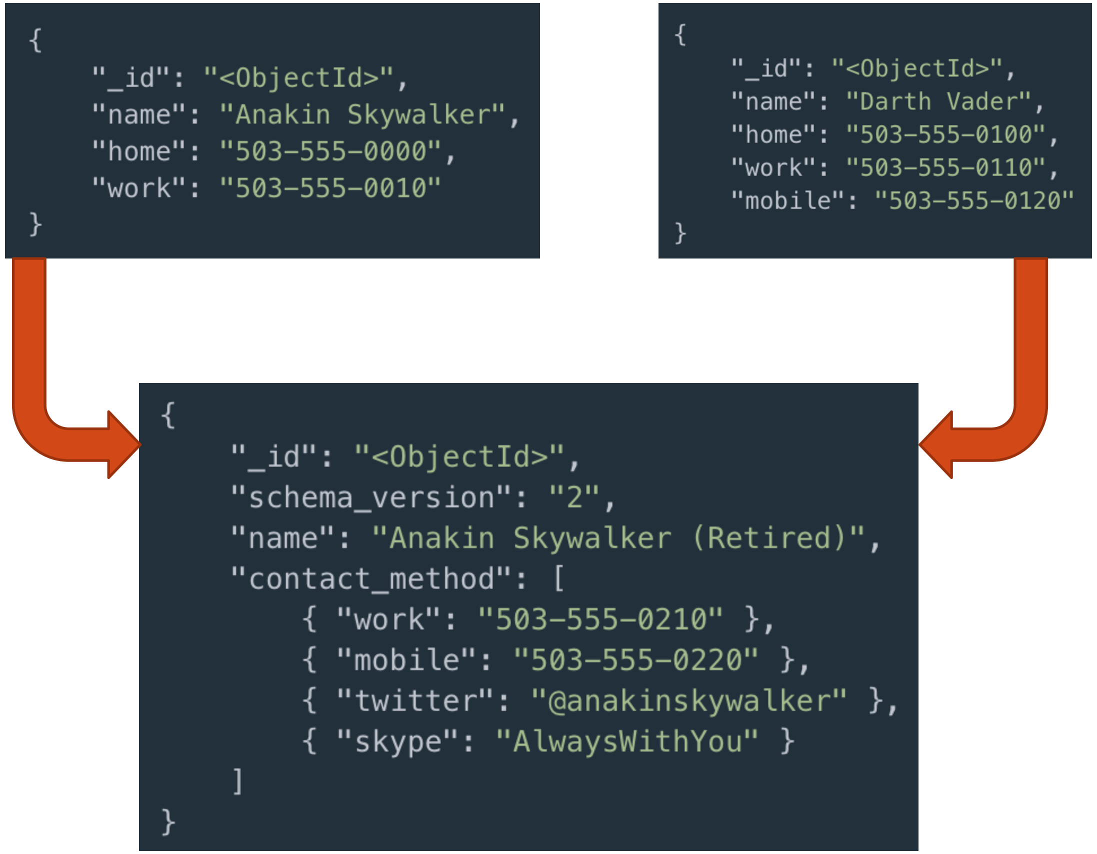
```
{
    "_id": "<ObjectId>",
    "name": "Darth Vader",
    "home": "503-555-0100",
    "work": "503-555-0110",
    "mobile": "503-555-0120"
}
```

```
{
    "_id": "<ObjectId>",
    "schema_version": "2",
    "name": "Anakin Skywalker (Retired)",
    "contact_method": [
        { "work": "503-555-0210" },
        { "mobile": "503-555-0220" },
        { "twitter": "@anakinskywalker" },
        { "skype": "AlwaysWithYou" }
    ]
}
```

- 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.

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



```
{
    "_id": "<ObjectId>",
    "name": "Anakin Skywalker",
    "home": "503-555-0000",
    "work": "503-555-0010"
}
```

```
{
    "_id": "<ObjectId>",
    "name": "Darth Vader",
    "home": "503-555-0100",
    "work": "503-555-0110",
    "mobile": "503-555-0120"
}
```

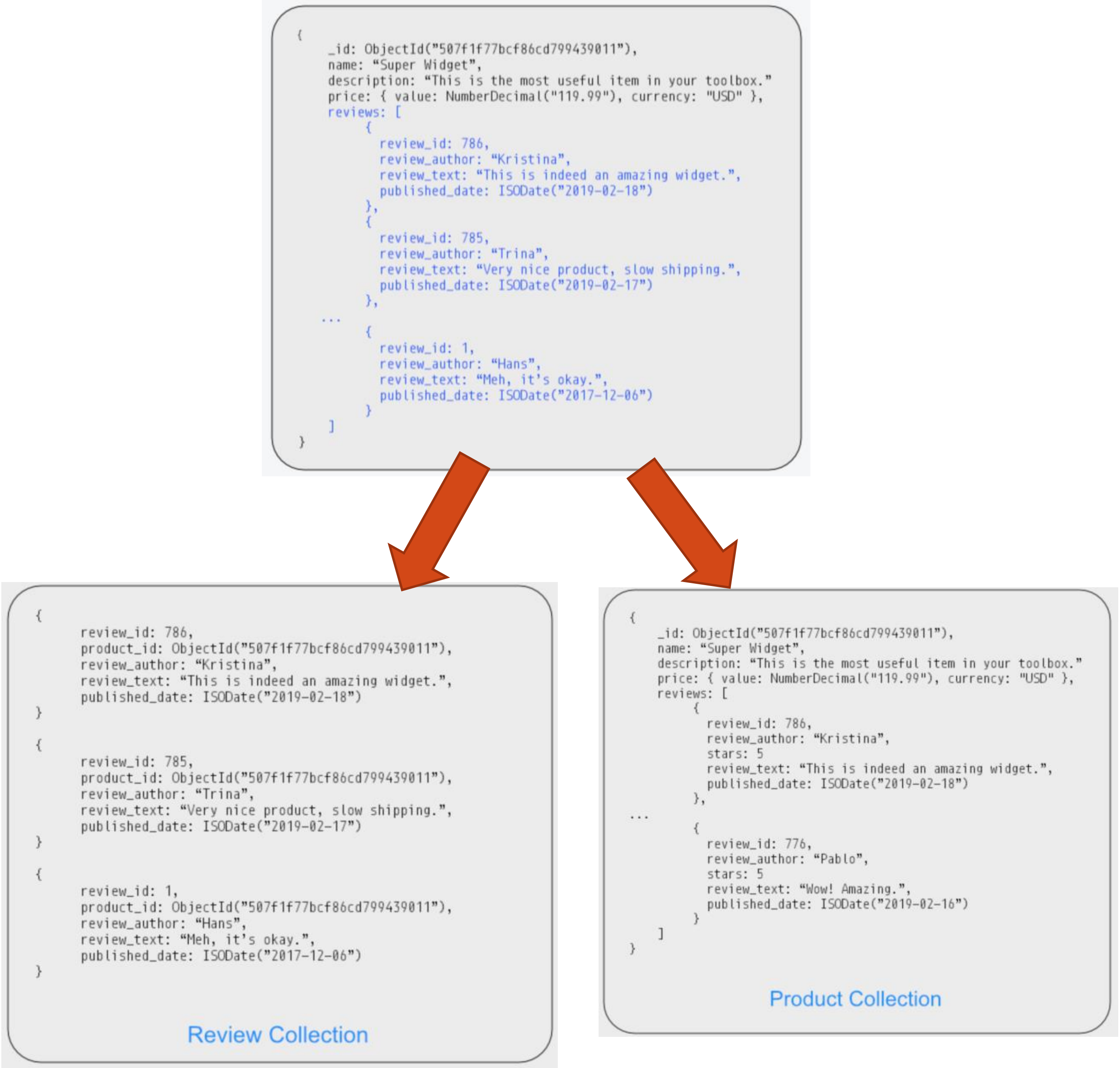
```
{
    "_id": "<ObjectId>",
    "schema_version": "2",
    "name": "Anakin Skywalker (Retired)",
    "contact_method": [
        { "work": "503-555-0210" },
        { "mobile": "503-555-0220" },
        { "twitter": "@anakinskywalker" },
        { "skype": "AlwaysWithYou" }
    ]
}
```

- 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

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



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

```
{
        review_id: 786,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Kristina”,
        review_text: “This is indeed an amazing widget.”,
        published_date: ISODate(”2019-02-18”)
}

{
        review_id: 785,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Trina”,
        review_text: “Very nice product, slow shipping.”,
        published_date: ISODate(”2019-02-17”)
}

{
        review_id: 1,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Hans”,
        review_text: “Meh, it’s okay.”,
        published_date: ISODate(”2017-12-06”)
}
```

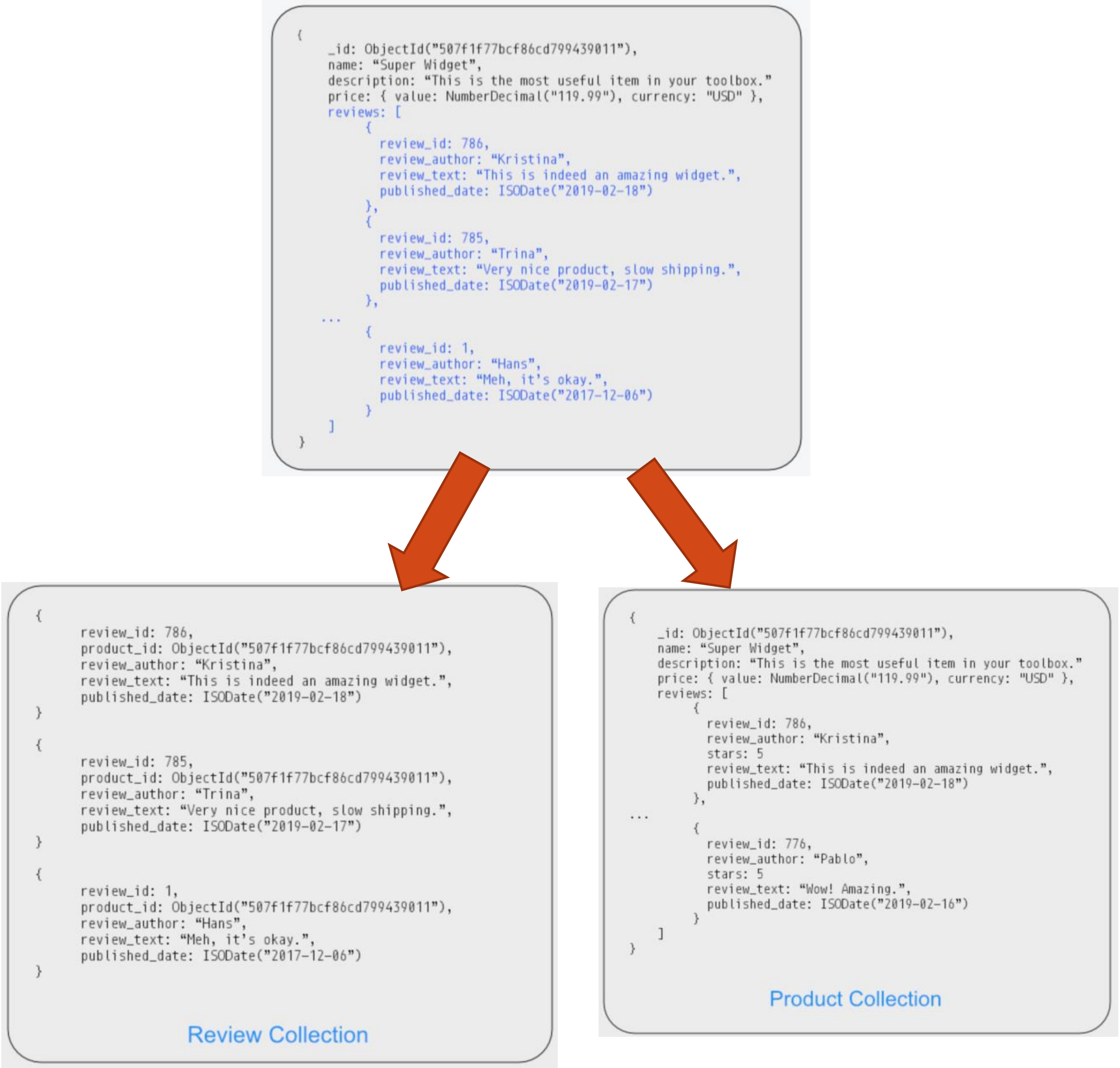
Review Collection

```
{
    _id: ObjectId(”507f1f77bcf86cd799439011”),
    name: “Super Widget”,
    description: “This is the most useful item in your toolbox.”
    price: { value: NumberDecimal("119.99"), currency: "USD" },
    reviews: [
          {
            review_id: 786,
            review_author: “Kristina”,
            stars: 5
            review_text: “This is indeed an amazing widget.”,
            published_date: ISODate(”2019-02-18”)
          },
...
          {
            review_id: 776,
            review_author: “Pablo”,
            stars: 5
            review_text: “Wow! Amazing.”,
            published_date: ISODate(”2019-02-16”)
          }
    ]
}
```

Product Collection

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.



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

```
{
        review_id: 786,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Kristina”,
        review_text: “This is indeed an amazing widget.”,
        published_date: ISODate(”2019-02-18”)
}

{
        review_id: 785,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Trina”,
        review_text: “Very nice product, slow shipping.”,
        published_date: ISODate(”2019-02-17”)
}

{
        review_id: 1,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Hans”,
        review_text: “Meh, it’s okay.”,
        published_date: ISODate(”2017-12-06”)
}
```

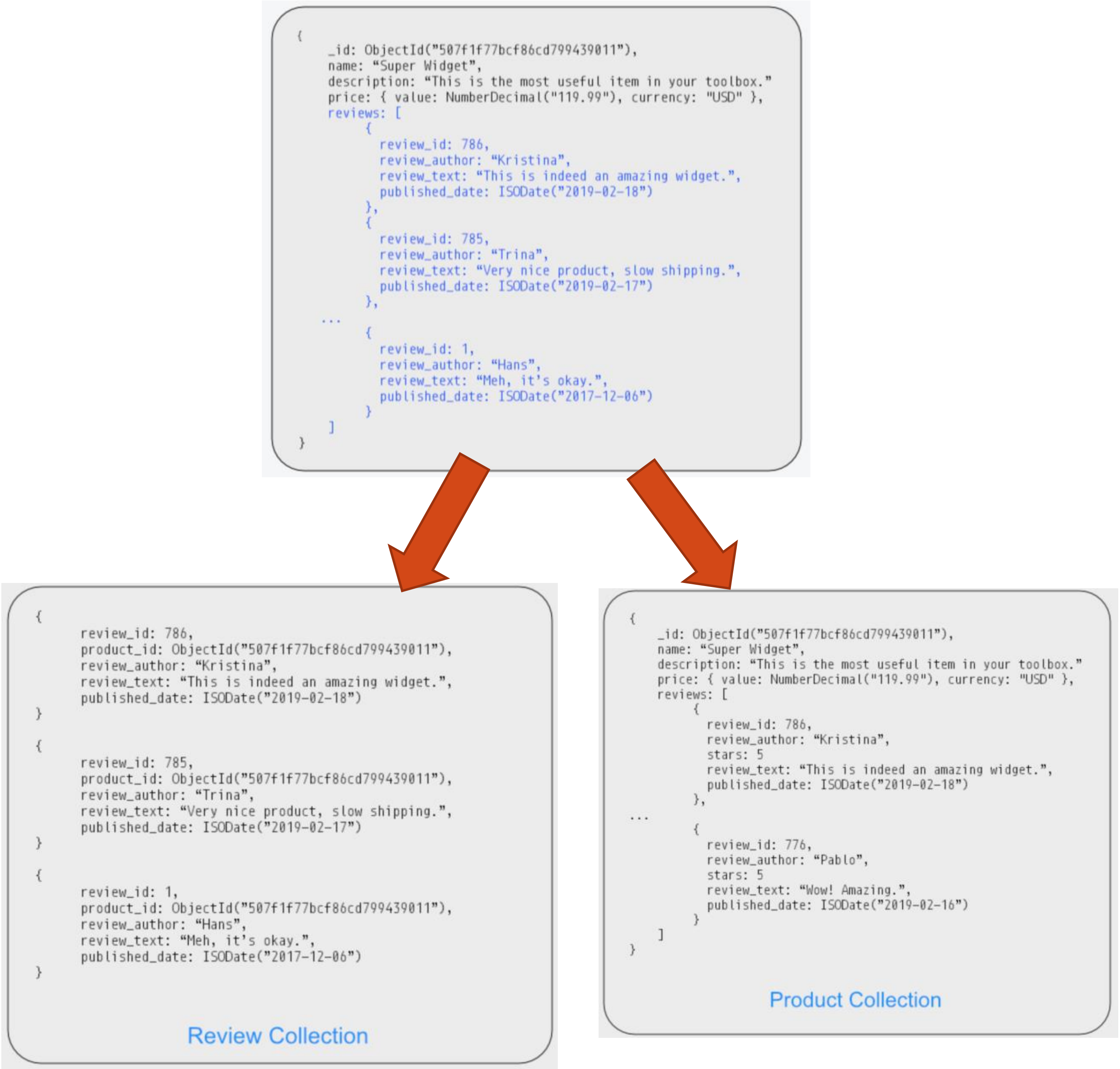
Review Collection

```
{
    _id: ObjectId(”507f1f77bcf86cd799439011”),
    name: “Super Widget”,
    description: “This is the most useful item in your toolbox.”
    price: { value: NumberDecimal("119.99"), currency: "USD" },
    reviews: [
          {
            review_id: 786,
            review_author: “Kristina”,
            stars: 5
            review_text: “This is indeed an amazing widget.”,
            published_date: ISODate(”2019-02-18”)
          },
...
          {
            review_id: 776,
            review_author: “Pablo”,
            stars: 5
            review_text: “Wow! Amazing.”,
            published_date: ISODate(”2019-02-16”)
          }
    ]
}
```

Product Collection

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



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

```
{
        review_id: 786,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Kristina”,
        review_text: “This is indeed an amazing widget.”,
        published_date: ISODate(”2019-02-18”)
}

{
        review_id: 785,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Trina”,
        review_text: “Very nice product, slow shipping.”,
        published_date: ISODate(”2019-02-17”)
}

{
        review_id: 1,
        product_id: ObjectId(”507f1f77bcf86cd799439011”),
        review_author: “Hans”,
        review_text: “Meh, it’s okay.”,
        published_date: ISODate(”2017-12-06”)
}
```

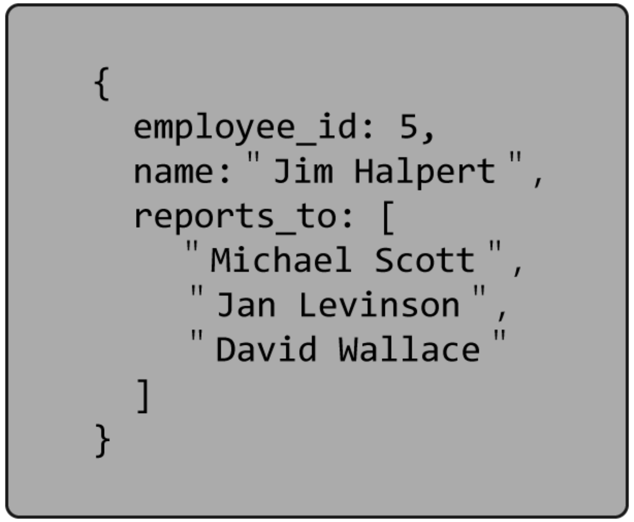
Review Collection

```
{
    _id: ObjectId(”507f1f77bcf86cd799439011”),
    name: “Super Widget”,
    description: “This is the most useful item in your toolbox.”
    price: { value: NumberDecimal("119.99"), currency: "USD" },
    reviews: [
          {
            review_id: 786,
            review_author: “Kristina”,
            stars: 5
            review_text: “This is indeed an amazing widget.”,
            published_date: ISODate(”2019-02-18”)
          },
...
          {
            review_id: 776,
            review_author: “Pablo”,
            stars: 5
            review_text: “Wow! Amazing.”,
            published_date: ISODate(”2019-02-16”)
          }
    ]
}
```

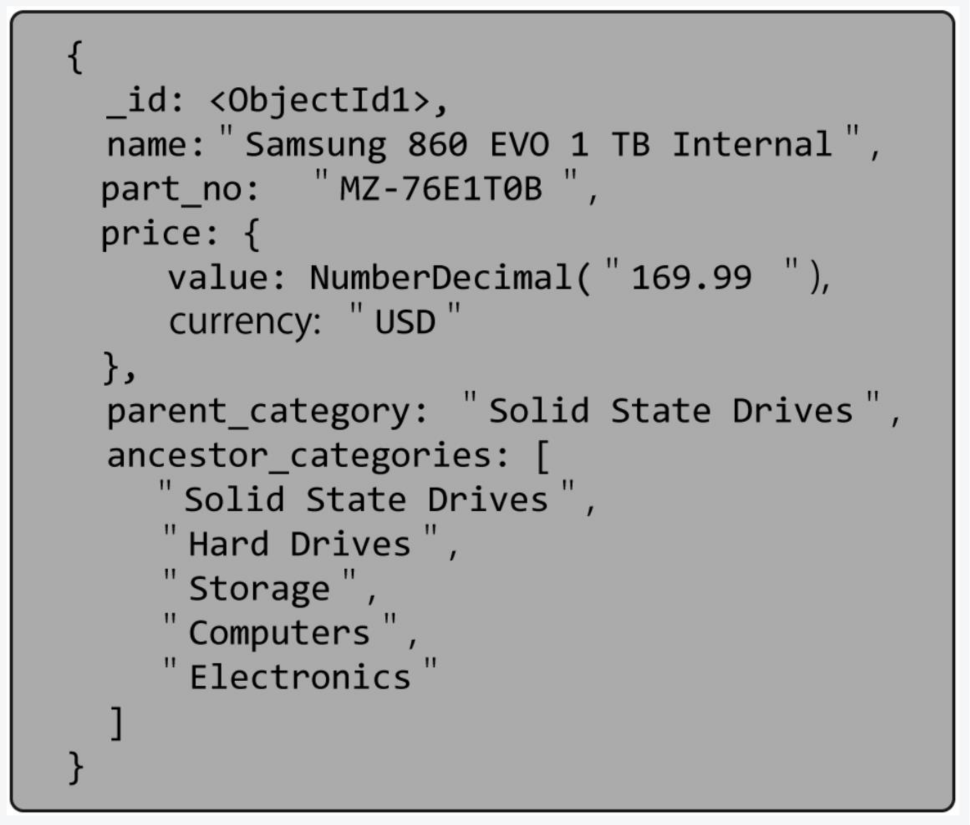
Product Collection

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 categories, which are part of other categories.



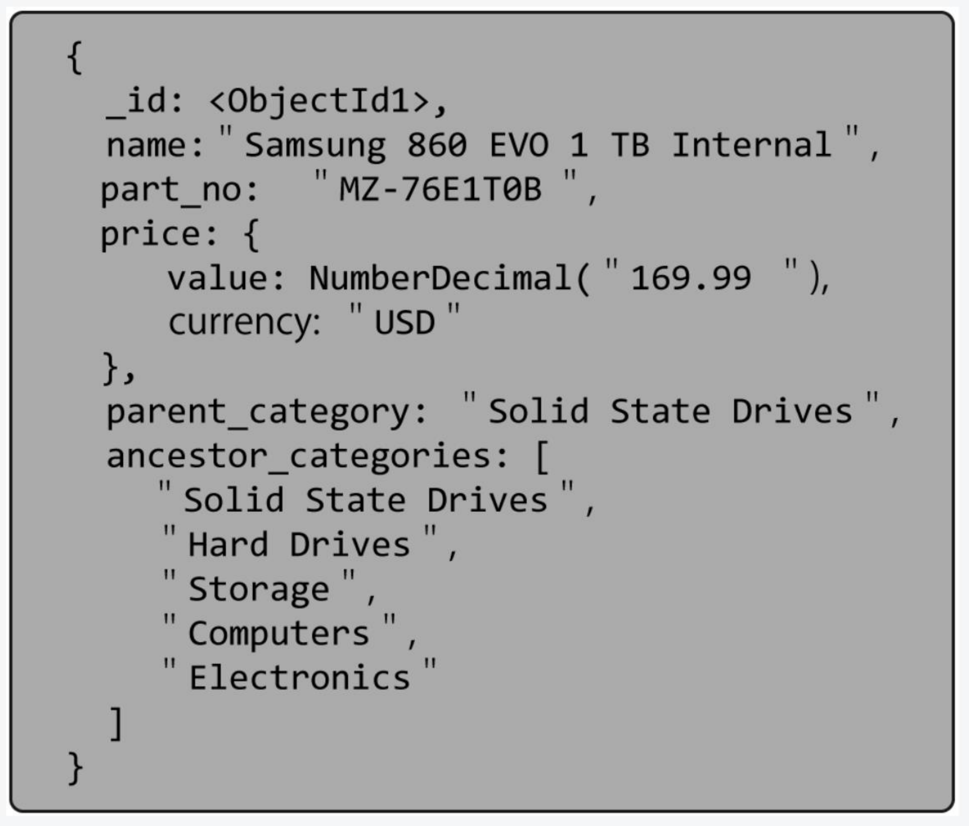
```
{
  employee_id: 5,
  name: "Jim Halpert",
  reports_to: [
     "Michael Scott",
     "Jan Levinson",
     "David Wallace"
  ]
}
```



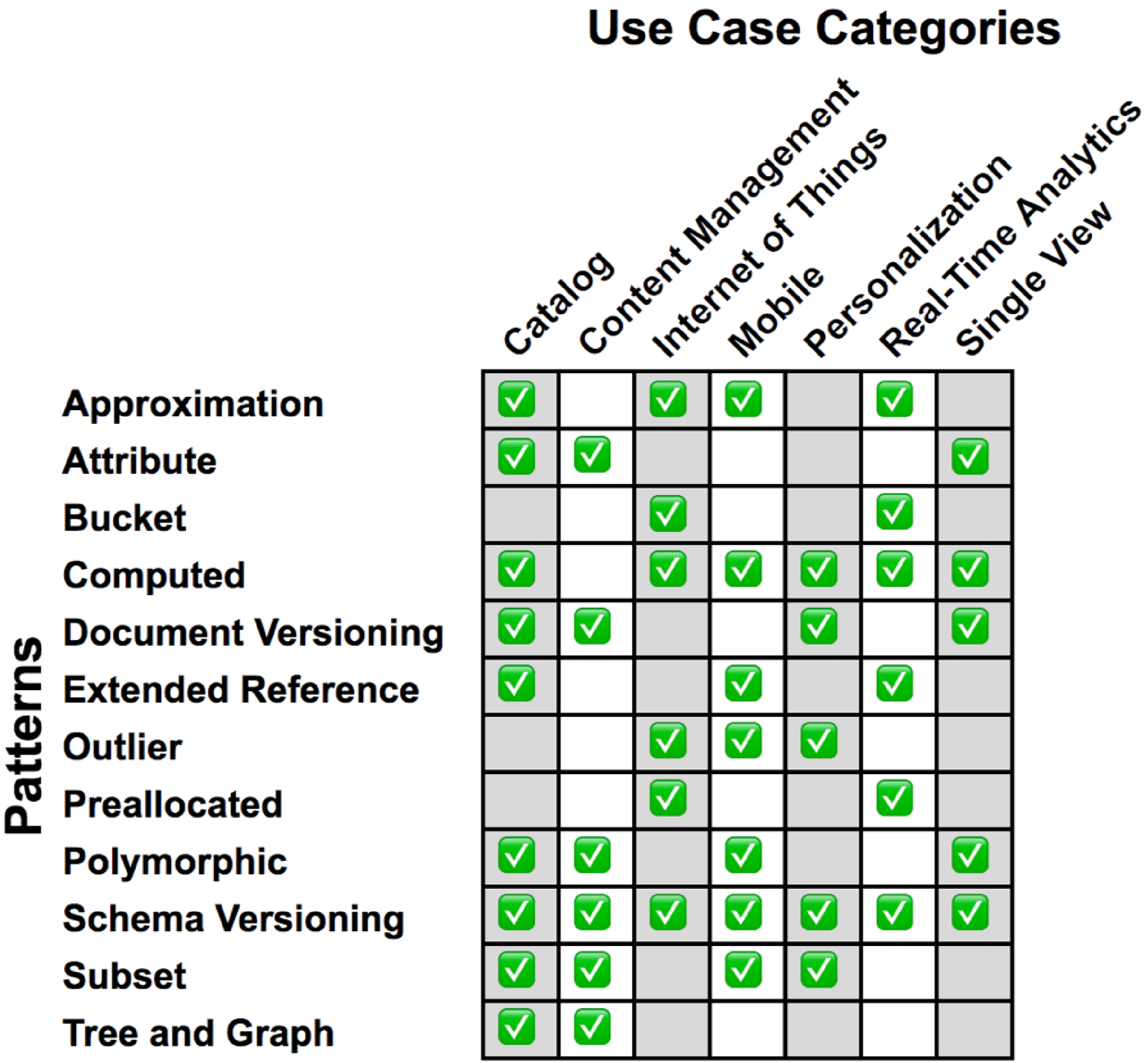
```
{
  _id: <ObjectId1>,
  name: "Samsung 860 EVO 1 TB Internal",
  part_no:   "MZ-76E1T0B ",
  price: {
     value: NumberDecimal("169.99  "),
     currency:  "USD"
  },
  parent_category:  "Solid State Drives",
  ancestor_categories: [
     "Solid State Drives",
     "Hard Drives",
     "Storage",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

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 "
  ]
}
```



Use Case Categories

Patterns	Catalog	Content Management	Internet of Things	Mobile	Personalization	Real-Time Analytics	Single View
Approximation	✓		✓	✓		✓	
Attribute	✓	✓					✓
Bucket			✓			✓	
Computed	✓		✓	✓	✓	✓	✓
Document Versioning	✓	✓			✓		✓
Extended Reference	✓			✓		✓	
Outlier			✓	✓	✓		
Preallocated			✓			✓	
Polymorphic	✓	✓		✓			✓
Schema Versioning	✓	✓	✓	✓	✓	✓	✓
Subset	✓	✓		✓	✓		
Tree and Graph	✓	✓					

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/