Data-driven systems

MongoDB

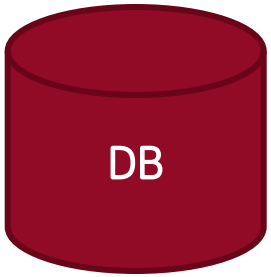
BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Contents

- „NoSQL”
- Basic concept
  - System architecture
  - Collection, document
  - Data representation
- CRUD operations, query
- Usage in .NET
- “Schema” design

NoSQL: motivation

Small database, simple schema



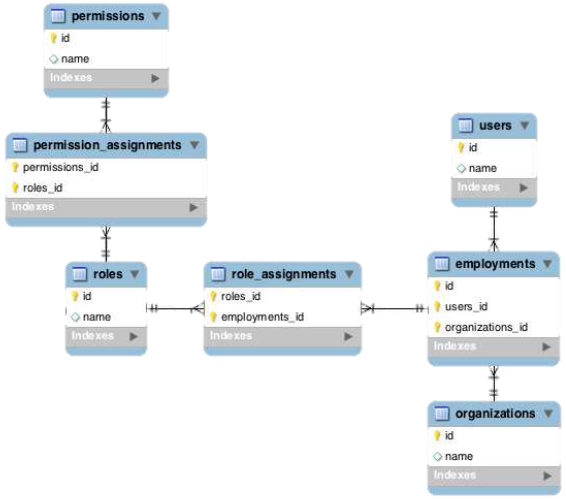
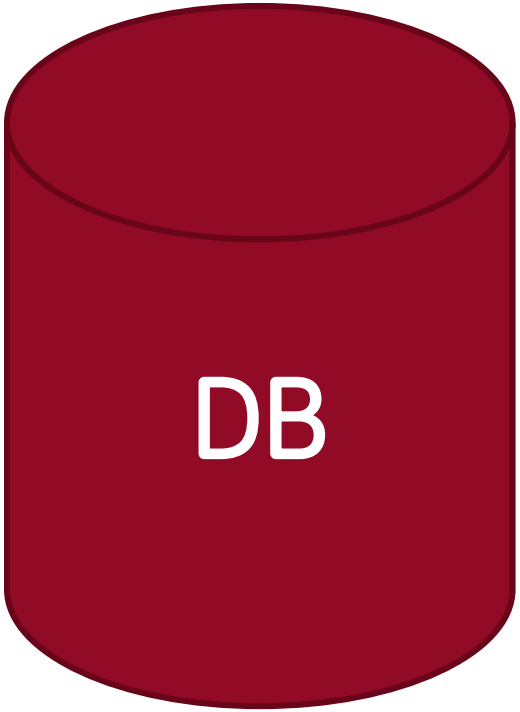


Image source: https://i.stack.imgur.com/cAuJ2.png

NoSQL: motivation

Application evolves -> schema gets complicated -> database grows



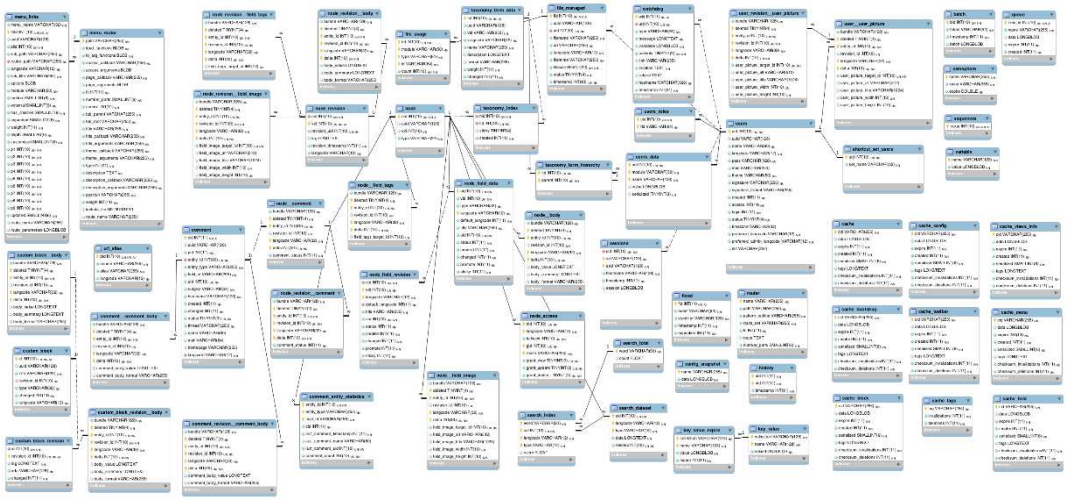


Image source:
https://www.activequerybuilder.com/blog/understanding-a-complex-database-structure/

NoSQL: motivation

Real life example

- Backend application developed for 10+ years
- 95+ tables
- Some tables having 50+ columns
- 5500 lines „create database.sql”

NoSQL: motivation

- Drawbacks when using relational databases
  - Schema is constantly changing
  - Data migration
  - Performance issues
    - Scalability is an issue due to strong consistency

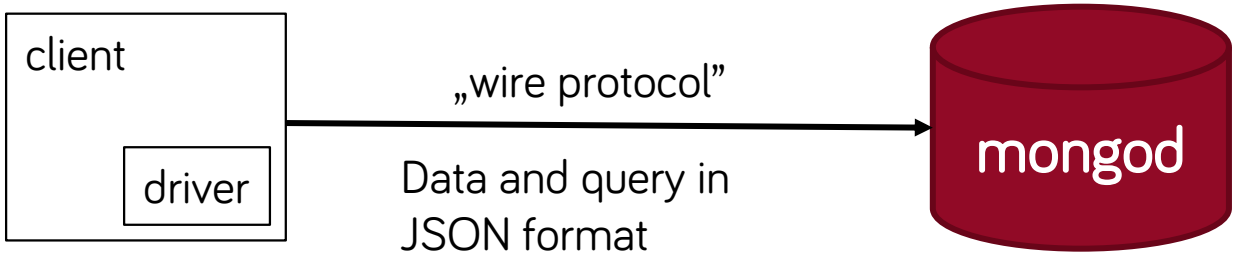
→Let us get rid of the strict schema: NoSQL

NoSQL: the name is misleading, has nothing to do with SQL

Basic concepts of MongoDB

Architecture

- System architecture



- Logical structure
  - Cluster
    - Server
      - Database
        - Collection
          - Document

Document

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

Document

- JSON/BSON
- Unit of storage
- Contains key-value pairs
- Value: string, number, null, date, binary, array, nested object, …
- Key
  - Free text
  - Unique name
  - Cannot begin with character $
  - _id implicit field
  - caseSensitive

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

- In the object-oriented world: an object
- Size limitation: 16MB

Collection

- Analogous to the table of a relational database
- There is no scheme, hence no need to define it
- Place of “similar” documents
- Indices are defined on collections
- Since there is no schema, there are no integrity requirements either

Collection

```
{
        name: "Sue",
        age: 26
}
{
        name: "John",
        age: 32,
        title: "CEO"
}
{
        name: "Ronald McDonald",
        age: "born in 1955"
}
```

Database

- Same purpose as in relational databases
- Umbrella over all data of an application
- Access rights is granted on the database level
- Name is case sensitive, typically used as lowercase

Relational schema / MongoDB

- Table
- Record
- Column
  - Scalar
- Integrity criteria
- Key
- Foreign keys
- Join
- Transactions

- Collection
- Document
- Field
  - Scalar & compound (nested)
- ~~Integrity criteria~~
- ObjectId, unique index
- Reference based on _id
- Nesting, arrays
- *“Transactions”*

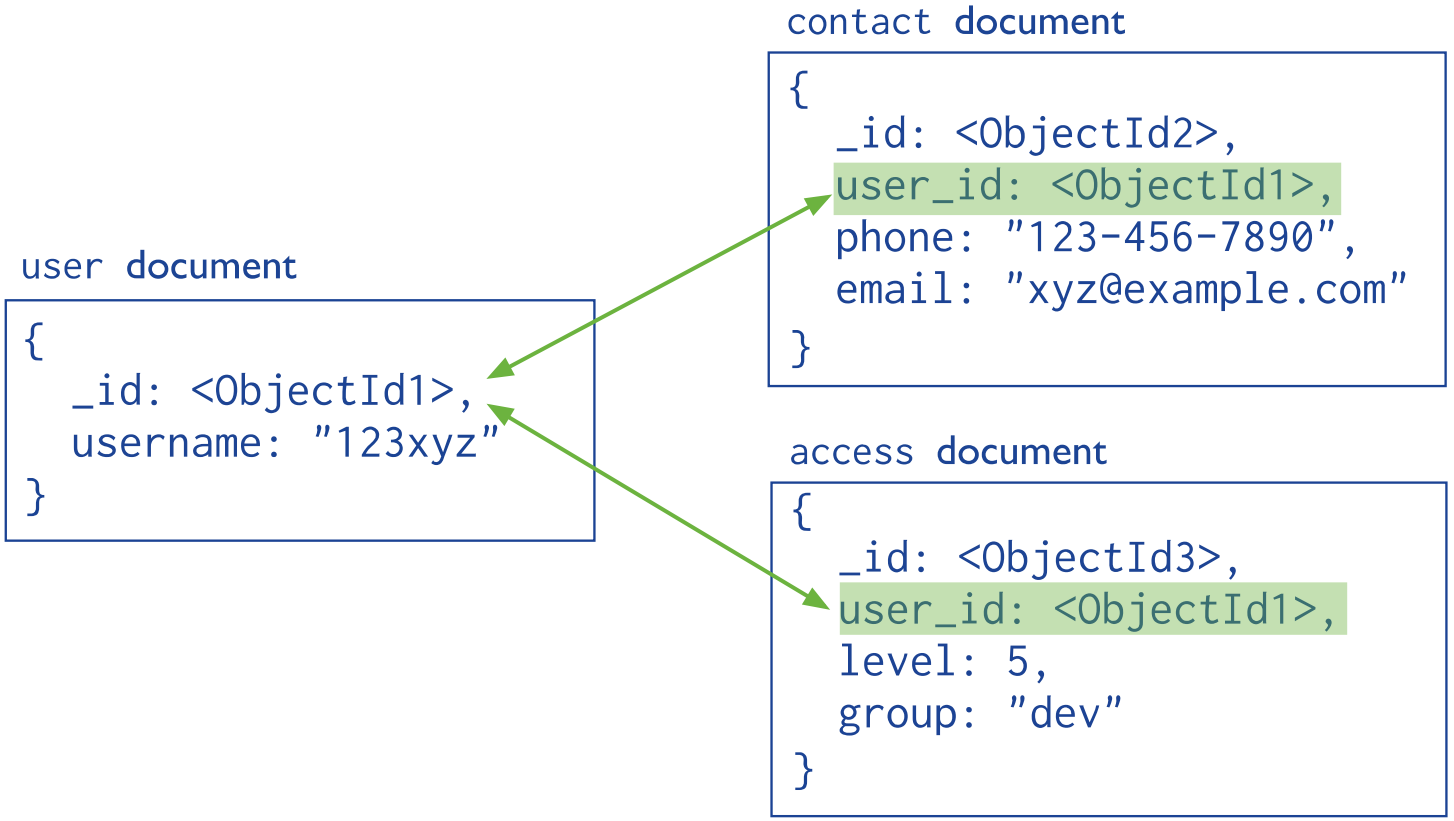
Key

- In every document: _id
- There are no keys otherwise
  - Cannot define a key
- ObjectId("507f191e810c19729de860ea")
  - Client driver or server generates it (not the application)
- 12 bytes
  - 4 bytes timestamp (seconds precision)
  - 5 bytes random (cluster node + mongod process id)
  - 3 bytes counter
- ~ globally unique

Key

- What the key is used for
  - Unambiguous identification → _id
  - Guarantee uniqueness → index
- There is no compound key
  - Compound indices instead

Reference to another document



Reference to another document

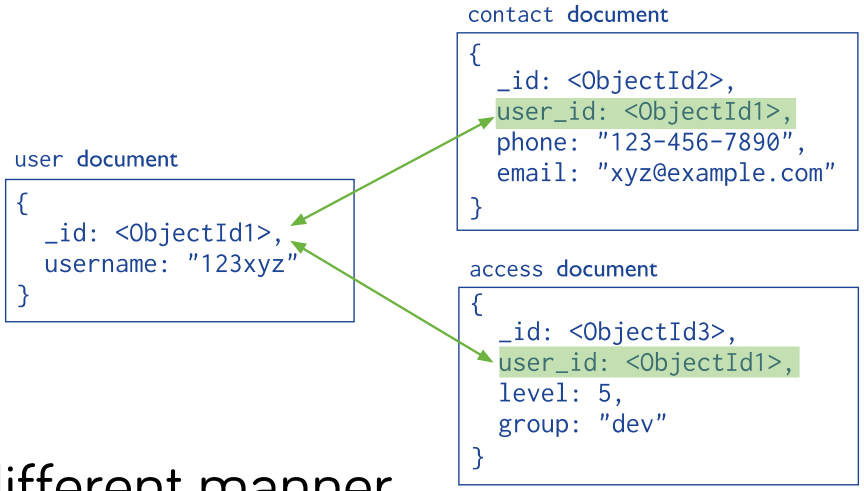
Similar to the object-oriented world

Denormalization!

Nesting, arrays

- Normalization, instead of join
- 1-1 connection
  - Nesting

- 1-many connection
  - Embedding as array
  - Or reference
    - There is ~~no~~ join, but in a different manner



Transactions

- Usually not supported in this world
- Supported since MongoDB 4.0 (not so long ago)
- ~~ACID~~
- Atomicity
- Instead of isolation: read/write concern
  - Not discussed more in this course.
  - https://docs.mongodb.com/manual/reference/read-concern/
  - https://docs.mongodb.com/manual/reference/write-concern/

MongoDB “schema” design

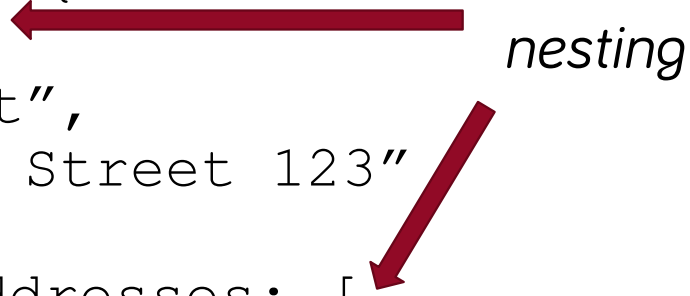
MongoDB “schema” design

- Collections usually yield themselves
- Prefer nesting to alternatives
  - But consider the document size limit!
- Transactions only when there is no other way

MongoDB schema design - example

- Customers

```
{
        _id: ...,
        name: ”John Doe”,
        email: john@doe.com,
        mainPlaceOfBusiness: {
              zipCode: 1234,
              city: ”Budapest”,
              street: ”Place Street 123”
        }
        additionalShippingAddresses: [
              { zip:..., city:..., street:... },
...
        ]
}
```

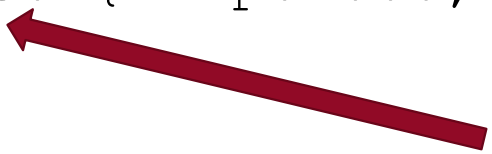


MongoDB schema design - example

- Order

```
{
        _id: ...,
        date: ...,
        customer: ObjectId(...),
        items: [
                { productId: 111, price: 222, piece: 3 },
                { productId: 111, price: 222, piece: 3 },
                ...
        ],
        shippingAddress: { zip: ..., city: ...}
}
```

reference



denormalization

MongoDB schema design - example

- Product

```
{
        _id: ...,
        name: Sofa,
        price: 1234,
        vatPercentage: 27,
        shippingInformation: {
              packagePieces: 2,
              packageSize: [
                    {width: 100, height: 45},
                    {width: 250, height: 35}
              ]
        }
}
```

denormalization

nesting

Transactions: SQL - NoSQL

- Typical SQL
  - ACID: Atomicity, Consistency, Isolation, Durability
  - If it were a single node
- Typical NoSQL
  - BASE: Basically Available, Soft State, Eventual Consistency
  - Giving up consistency for availability
  - Scenario: e.g. number of Facebook likes, name change, etc.

Transactions in MongoDB

- Updating a document is atomic
- It supports distributed transactions
  - Even with ACID properties
  - Performance degrades significantly!
- Isolation levels "instead" of concerns :
  - https://docs.mongodb.com/manual/reference/read-concern/
  - https://docs.mongodb.com/manual/reference/write-concern/

SQL vs NoSQL

- SQL challenges
  - Size / rigid data model / **scaling** / **availability**
- DB-level solutions introduced by NoSQL can present new challenges in the other layers
  - Denormalization -> consistency maintenance
  - Scaling -> transactional constraints
  - Semi-structured data -> Type information is lost
    - How is data displayed on the user interface?
    - What data does the API provide to additional layers and external services?
    - How do we match the code to several scheme versions?

MongoDB CRUD operations in .NET

MongoDB protocol

- „Wire protocol”: TCP/IP based binary
- Client applications, drivers for popular platforms
  - https://docs.mongodb.com/ecosystem/drivers/
- Both request and response are JSON documents
  - It should be thought of like the SQL language, only it is platform specific
  - Certain drivers provide this level
  - Driver compatible with more advanced languages (e.g. C#).

MongoDB "raw" language

```
db.inventory.find( {
    $and: [
        {
            price: { $ne: 1.99 }
        },
        {
            price: { $exists: true}
        }
    ]
})
```

- For simplicity, we will NOT show the "raw" JSON queries, only the .NET API

MongoDB .NET

- MongoDB.Driver
- https://www.nuget.org/packages/mongodb.driver

→ Check the lecture notes

Code – database mapping

- Code generation
  - > Based on schema – if exists
  - > Based on sample jsons
- Manually – code first

```
public class Product
{
    public ObjectId Id { get; set; } // ez lesz az elsődleges kulcs helyett az _id
    public string Name { get; set; }
    public float Price { get; set; }
    public int Stock { get; set; }
    public string[] Categories { get; set; } // tömb értékű mező
    public VAT VAT { get; set; } // beágyazást alkalmazunk
}

public class VAT // mivel ez beágyazott entitás, így nem adunk neki egyedi azonosí
{
    public string VATCategoryName { get; set; }
    public float Percentage { get; set; }
}
```

Customization

- Pascalcasing is converted by the built-in convention
  - > We can also create our own conventions
- Customization with attributes

```
public class Product
{
    // _id mezőre képződik le
    [BsonId]
    public ObjectId Azonosito { get; set; }

    // megadhatjuk a MongoDB dokumentumban használatos nevet
    [BsonElement("price")]
    public string Ar { get; set; }

    // kihagyhatunk egyes mezőket
    [BsonIgnore]
    public string NemMentett { get; set; }
}
```

Queries

- We perform operations on collections

```
var collection = db.GetCollection<Product>("products");
```

- Find
  - A lambda expression

```
collection.Find(x => x.Price < 123 && x.Name.Contains("red"));
```

  - Raw MongoDB query in its own json language

```
collection.Find(
    Builders<Product>.Filter.And(
        Builders<Product>.Filter.Lt(x => x.Price, 123),
        Builders<Product>.Filter.Regex(x => x.Name, "/red/s"
    )
);
```

```
{
  "price": {
    "$lt": 123.0
  },
  "name": "/red/s"
}
```

Operators

- Filters only apply to constant values

```
collection.Find(x => x.Price != 123);
collection.Find(Builders<Product>.Filter.Ne(x => x.Price, 123));
```

- Null / non-existent key

```
collection.Find(x => x.VAT != null);
collection.Find(Builders<Product>.Filter.Exists(x => x.VAT));
```

- Array field filtering

```
// azon termékeket, amelyek a jelzett kategóriában vannak
collection.Find(Builders<Product>.Filter.AnyEq(x => x.Categories, "Labdák"));
```

- Sorting, pagination

```
collection.Find(...)
    .Sort(Builders<Product>.Sort.Ascending(x => x.Name))
    .Skip(100).Limit(100);
```

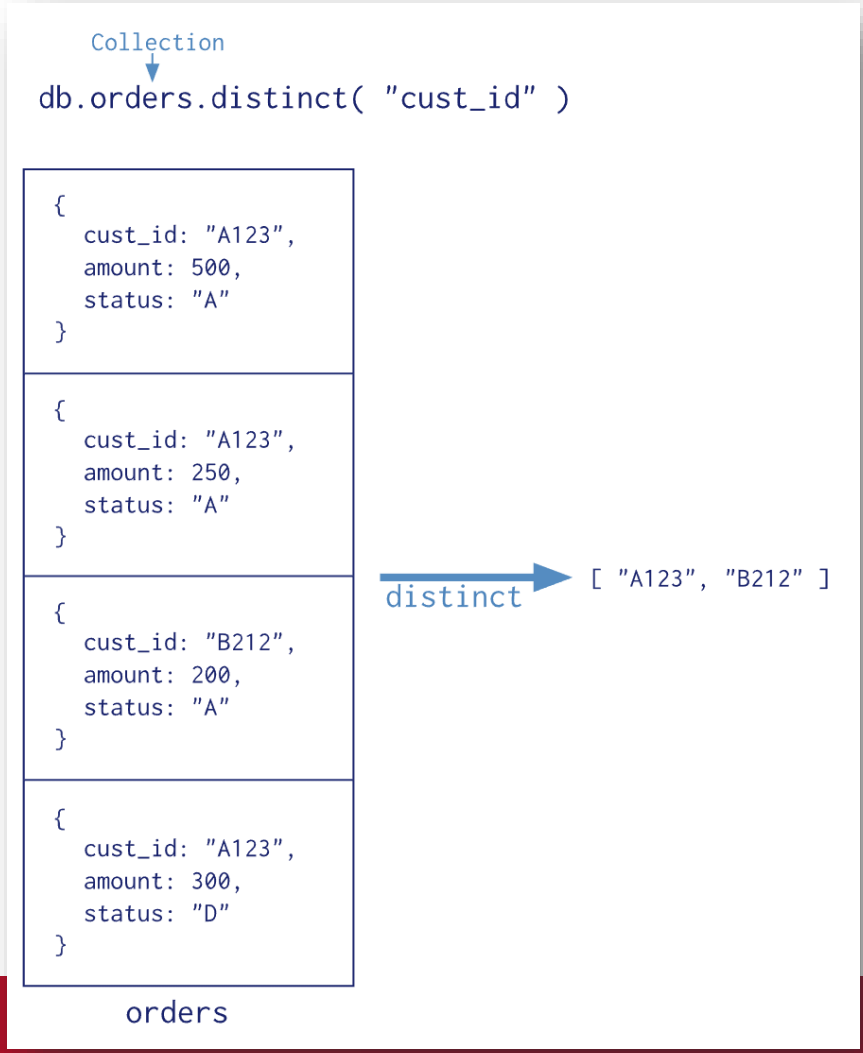
Result processing

- Listing
  - `. ToList()`
- Query one element
  - `. First() / FirstOrDefault() / Single() / …`
- Cursor
  - We process the result continuously

```
var cur = collection.Find(...).ToCursor();
while (cur.MoveNext()) // kurzor léptetése
{
    foreach (var t in cur.Current) // a kurzor aktuális eleme nem egy dokumentum
    { ... }
}
```

Aggregation pipeline

- Processing multiple documents on the server side
- Single Purpose
  - distinct
  - count



Generic aggregation

- Several actions can be defined one after the other
  - filtering, grouping, counting, …

```
// A "Labdák" kategóriába tartozó termékek
// ÁFA kulcs szerint csoportosítva
foreach (var g in collection.Aggregate()
                            // szűrés
                            .Match(Builders<Product>
                                .Filter.AnyEq(x => x.Categories, "Labdák"))
                            // csoportosítás
                            .Group(x => x.VAT.Percentage, x => x)
                            .ToList())
{
    Console.WriteLine($"VAT percentage: {g.Key}");
    foreach (var p in g)
        Console.WriteLine($"\tProduct: {p.Name}");
}
```

"Join" in mongodb

- Left Outer Join can be requested in an aggregation operation with the $lookup function
  - You can even join array elements
- Mongodb is not made for this
- If we were to use too many lookups, think again
  - Scheme
  - Is this the right tool for the job?

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

Insert

- Simple object instantiation
- We do not fill in the ID
- **InsertOne / InsertMany**

```
var newProduct = new Product
{
    Name = "Alma",
    Price = 890,
    Categories = new[] { "Gyümölcsök" }
};
collection.InsertOne(newProduct);

Console.WriteLine($"Beszúrt rekord id: {newProduct.Id}");
```

Delete

- Delete based on a filter condition
  - One or more documents
  - **DeleteOne / DeleteMany**

```
var deleteResult = collection.DeleteOne(x => x.Id == new ObjectId("..."));
Console.WriteLine($"Törölve: {deleteResult.DeletedCount} db");
```

Replace entire document

- Replace a document based on filtering
  - We can even replace the ID
  - `ReplaceOne / FindOneAndReplace`

```
var replacementProduct = new Product
{
    Name = "Alma",
    Price = 890,
    Categories = new[] { "Gyümölcsök" }
};
var replaceResult = collection.ReplaceOne(
    x => x.Id == new ObjectId("..."), replacementProduct
);

Console.WriteLine($"Módosítva: {replaceResult.ModifiedCount}");
```

Modify fields

- We modify the specific fields of the retrieved document
  - `UpdateOne / UpdateMany`

```
collection.UpdateOne(
    filter: x => x.Id == new ObjectId("..."),
    update: Builders<Product>.Update.Set(x => x.Stock, 5));
```

```
collection.UpdateOne(
    filter: x => x.Id == new ObjectId("..."),
    update: Builders<Product>.Update
        // raktárkészlet legyen 5
        .Set(x => x.Stock, 5)
        // mai dátumot beírjuk, mint a frissítés ideje
        .CurrentDate(x => x.StockUpdated)
        // töröljük a frissítendő jelzést
        .Unset(x => x.NeedsUpdate)
);
```

Insert or modify

- Upsert (update / insert) operation
- We either have it or create it
  - > Modifications can also be made in this way
- Useful for concurrent access

```
collection.ReplaceOne(
    filter: x => x.Id == new ObjectId("..."),
    replacement: replacementObject,
    options: new UpdateOptions() { IsUpsert = true });
```

Field operations

- Set: setting the field value;
- SetOnInsert: like Set, but only executed when a new document is inserted;
- Unset: delete a field (remove the key and value from the document);
- CurrentDate: enter the current date;
- Inc: increase value;
- Min, Max: replacement of the field value, if the specified value is lower/higher than the current value of the field;
- Mul: multiplication of value;
- PopFirst, PopLast: remove first/last element from array;
- Pull: removing a value from an array;
- Push: adding a value to an array at the end (more options in the same operator: sorting an array, keeping the first n elements of an array);
- AddToSet: add a value to an array if it does not already exist.