Praca z danymi

Operacje CRUD, potok agregacji, operatory wyrażeń i indeksy.

- Operacje na dokumentach
- Aggregation pipeline
- Operatory wyrażeń (logika i formatowanie wewnątrz etapów)
- Indeksy w MongoDB

Operacje na dokumentach

Poziom: podstawy **Dotyczy: MongoDB 5.0–8.0** **Czas: ~20 min**

Wstawianie, wyszukiwanie, aktualizowanie i usuwanie dokumentów — cztery operacje, na których stoi cała praca z danymi w MongoDB, wykonywane wprost z powłoki `mongosh`.

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
B2B	business to business: sprzedaż i systemy skierowane do innych firm, nie do klientów indywidualnych
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

Komendy wykonujemy w powłoce `mongosh`, opisanej na stronie mongosh: powłoka administracyjna, po wcześniejszym przełączeniu się na właściwą bazę poleceniem `use`.

TYLKO ODCZYT

Wstawienie pojedynczego dokumentu: insertOne

```
db.uzytkownicy.insertOne({ imie: "Anna", wiek: 30, miasto: "Warszawa" })
```

Wstawiamy jeden dokument do kolekcji. Jeśli kolekcja jeszcze nie istnieje, MongoDB tworzy ją automatycznie przy pierwszym zapisie.

Przykład:

```
{
  acknowledged: true,
  insertedId: ObjectId('66f1b2c3d4e5f6a7b8c9d0e1')
}
```

TYLKO ODCZYT

Wstawienie wielu dokumentów naraz: insertMany

```
db.uzytkownicy.insertMany([
  { imie: "Jan", wiek: 41 },
  { imie: "Ola", wiek: 22 }
])
```

Wstawiamy tablicę dokumentów jednym poleceniem — szybsze niż wywoływanie `insertOne` w pętli, bo wymaga jednej podróży do serwera zamiast wielu.

Przykład:

```
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('66f1b30ad4e5f6a7b8c9d0e2'),
    '1': ObjectId('66f1b30ad4e5f6a7b8c9d0e3')
  }
}
```

TYLKO ODCZYT

Wyszukanie wszystkich dokumentów w kolekcji: find

```
db.uzytkownicy.find()
```

Zwracamy wszystkie dokumenty kolekcji bez żadnego filtra. Wynik w powłoce jest domyślnie sformatowany czytelnie, po jednym dokumencie na wpis.

Przykład:

```
[
  { _id: ObjectId('66f1b2c3d4e5f6a7b8c9d0e1'), imie: 'Anna', wiek: 30, miasto:
'Warszawa' },
  { _id: ObjectId('66f1b30ad4e5f6a7b8c9d0e2'), imie: 'Jan', wiek: 41 },
  { _id: ObjectId('66f1b30ad4e5f6a7b8c9d0e3'), imie: 'Ola', wiek: 22 }
]
```

TYLKO ODCZYT

Wyszukanie z filtrem: operator $eq

```
db.uzytkownicy.find({ imie: "Anna" })
```

Dopasowanie wartości pola wprost, bez operatora, jest równoważne użyciu operatora `$eq` — obie formy zwracają ten sam wynik, ale forma bez operatora jest krótsza i częściej spotykana w praktyce.

TYLKO ODCZYT

Wyszukanie z operatorem porównania: $gt

```
db.uzytkownicy.find({ wiek: { $gt: 25 } })
```

Zwracamy dokumenty, w których wartość pola `wiek` jest większa niż podana liczba. Operator zapisujemy jako pole wewnątrz obiektu przypisanego do filtrowanego klucza — to jest forma zapytania, różna składniowo od formy używanej wewnątrz wyrażeń potoku agregacji, opisanej na stronie Operatory wyrażeń.

Przykład:

```
[
  { _id: ObjectId('66f1b2c3d4e5f6a7b8c9d0e1'), imie: 'Anna', wiek: 30, miasto:
'Warszawa' },
  { _id: ObjectId('66f1b30ad4e5f6a7b8c9d0e2'), imie: 'Jan', wiek: 41 }
]
```

TYLKO ODCZYT

Wyszukanie z listą dopuszczalnych wartości: $in

```
db.uzytkownicy.find({ imie: { $in: ["Anna", "Jan"] } })
```

Zwracamy dokumenty, w których wartość pola pasuje do dowolnej z wartości podanych w tablicy — odpowiednik operatora `IN` z SQL.

SQL = Structured Query Language: język zapytań do relacyjnych baz danych

TYLKO ODCZYT

Wyszukanie dopasowaniem tekstowym: $regex

```
db.uzytkownicy.find({ imie: { $regex: "^An", $options: "i" } })
```

Dopasowujemy wartość pola tekstowego do wyrażenia regularnego. Opcja `i` w `$options` sprawia, że dopasowanie ignoruje wielkość liter.

TYLKO ODCZYT

Wyszukanie po obecności pola: $exists

```
db.uzytkownicy.find({ miasto: { $exists: true } })
```

Zwracamy dokumenty, w których wskazane pole w ogóle występuje, niezależnie od jego wartości — przydatne w kolekcjach bez wymuszonego schematu, gdzie część dokumentów może w ogóle nie mieć danego pola, zamiast mieć je ustawione na `null`.

TYLKO ODCZYT

Złożenie warunków spójnikiem logicznym: $and i $or

```
db.uzytkownicy.find({ $and: [ { wiek: { $gte: 18 } }, { miasto: "Warszawa" } ] })
```

Łączymy kilka warunków spójnikiem logicznym `AND` — w praktyce rzadko potrzebny jawnie, bo kilka pól podanych obok siebie w jednym obiekcie filtra jest już domyślnie łączonych przez `AND`. Jawny `$and` jest konieczny, gdy ten sam klucz musi wystąpić w filtrze więcej niż raz.

```
db.uzytkownicy.find({ $or: [ { wiek: { $lt: 18 } }, { miasto: "Kraków" } ] })
```

Odpowiednik spójnika `OR` — dokument spełnia warunek, jeśli pasuje do choćby jednego z podanych.

TYLKO ODCZYT

Ograniczenie zwracanych pól: projekcja

```
db.uzytkownicy.find({ miasto: "Warszawa" }, { imie: 1, wiek: 1, _id: 0 })
```

Drugi argument `find()` to projekcja — decyduje, które pola trafią do wyniku. Wartość `1` oznacza pokazanie pola, `0` jego ukrycie; pole `_id` jest jedynym, które można wyłączyć razem z jawnie włączonymi innymi polami.

Przykład:

```
[ { imie: 'Anna', wiek: 30 } ]
```

TYLKO ODCZYT

Aktualizacja pierwszego pasującego dokumentu: updateOne

```
db.uzytkownicy.updateOne(
  { imie: "Anna" },
  { $set: { wiek: 31 } }
)
```

Aktualizujemy pierwszy dokument pasujący do filtra. Operator `$set` ustawia wartość wskazanego pola, tworząc je, jeśli wcześniej nie istniało.

Przykład:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1
}
```

TYLKO ODCZYT

Zwiększenie wartości liczbowej: $inc

```
db.uzytkownicy.updateOne({ imie: "Anna" }, { $inc: { wiek: 1 } })
```

Zwiększamy (albo zmniejszamy, przy wartości ujemnej) liczbową wartość pola o podaną wielkość — bez konieczności najpierw odczytywać bieżącej wartości po stronie aplikacji.

🔒 TYLKO ODCZYT

Dodanie elementu do tablicy: $push

```
db.uzytkownicy.updateOne({ imie: "Anna" }, { $push: { role: "operator" } })
```

Dopisujemy nową wartość na koniec tablicy przechowywanej w polu dokumentu. Jeśli pole jeszcze nie istnieje, zostaje utworzone jako nowa tablica z jednym elementem.

🔒 TYLKO ODCZYT

Usunięcie pola z dokumentu: $unset

```
db.uzytkownicy.updateOne({ imie: "Anna" }, { $unset: { miasto: "" } })
```

Usuwamy wskazane pole z dokumentu całkowicie — wartość podana po prawej stronie nie ma znaczenia, liczy się tylko sama nazwa pola.

🔒 TYLKO ODCZYT

Aktualizacja wielu dokumentów naraz: updateMany

```
db.uzytkownicy.updateMany(
  { wiek: { $lt: 18 } },
  { $set: { pelnoletni: false } }
)
```

Aktualizujemy wszystkie dokumenty pasujące do filtra, nie tylko pierwszy — składnia operatorów aktualizacji jest identyczna jak w `updateOne`.

🔒 TYLKO ODCZYT

Aktualizacja z utworzeniem dokumentu, gdy brak dopasowania: upsert

```
db.uzytkownicy.updateOne(
  { imie: "Marek" },
  { $set: { wiek: 27 } },
```

```
  { upsert: true }
)
```

Trzeci argument `{ upsert: true }` sprawia, że jeśli żaden dokument nie pasuje do filtra, MongoDB tworzy nowy dokument łączący pola z filtra i z operatora `$set`, zamiast nic nie robić.

□□ TYLKO ODCZYT

Zastąpienie całego dokumentu: replaceOne

```
db.uzytkownicy.replaceOne(
  { imie: "Jan" },
  { imie: "Jan", wiek: 42, miasto: "Poznań" }
)
```

Zastępujemy cały dokument nowym, podanym wprost — w odróżnieniu od `updateOne` z `$set`, pola nieobecne w nowym dokumencie znikają, bo cały poprzedni dokument zostaje nadpisany, nie scalony.

□□ TYLKO ODCZYT

Usunięcie pierwszego pasującego dokumentu: deleteOne

```
db.uzytkownicy.deleteOne({ imie: "Ola" })
```

Usuwamy pierwszy dokument pasujący do filtra.

Przykład:

```
{ acknowledged: true, deletedCount: 1 }
```

□□ TYLKO ODCZYT

Usunięcie wszystkich pasujących dokumentów: deleteMany

```
db.uzytkownicy.deleteMany({ wiek: { $lt: 18 } })
```

Usuwamy wszystkie dokumenty pasujące do filtra. Pusty filtr `{}` usunąłby całą zawartość kolekcji naraz, dlatego warto zawsze świadomie sprawdzić filtr, zanim wykonamy tę operację na danych produkcyjnych.

TYLKO ODCZYT

Sortowanie, stronicowanie i ograniczenie wyniku: sort, skip, limit

```
db.uzytkownicy.find().sort({ wiek: -1 }).skip(10).limit(5)
```

Sortujemy wynik malejąco po wieku (`-1`, rosnąco byłoby `1`), pomijamy pierwsze dziesięć dopasowań i zwracamy kolejnych pięć — typowy wzorzec stronicowania wyników w interfejsie użytkownika.

TYLKO ODCZYT

Policzenie dokumentów pasujących do filtra: countDocuments

```
db.uzytkownicy.countDocuments({ miasto: "Warszawa" })
```

Zwracamy liczbę dokumentów pasujących do filtra, bez pobierania samych dokumentów — szybsze niż pobranie wyniku i policzenie go po stronie aplikacji.

Przykład:

```
3
```

TYLKO ODCZYT

Pobranie unikalnych wartości pola: distinct

```
db.uzytkownicy.distinct("miasto")
```

Zwracamy listę unikalnych wartości, jakie występują we wskazanym polu w całej kolekcji — odpowiednik `SELECT DISTINCT miasto FROM uzytkownicy` w bazie relacyjnej.

Przykład:

```
[ 'Warszawa', 'Poznań', 'Kraków' ]
```

Z życia wzięte

> **Z ŻYCIA WZIĘTE**
>
> Kasia chce wycofać z katalogu jedną nieaktualną promocję i uruchamia updateMany z warunkiem, który miał dotyczyć tylko tej promocji, ale przez literówkę w nazwie pola pasuje do wszystkich produktów w kolekcji. W ciągu sekundy wszystkie produkty w sklepie B2B tracą pole cenaPromocyjna, co Marek zgłasza dopiero po południu, gdy klienci przestają widzieć promocje na stronie. Odtwarzamy poprawne wartości z ostatniego backupu opisanego na stronie Backup i odtwarzanie, a Kasia uczy się testować warunek find przed uruchomieniem update na całej kolekcji.

Częste błędy

Mylenie składni operatora zapytania z operatorem wyrażenia. `{ wiek: { $gt: 25 } }` działa w `find()`, ale wewnątrz potoku agregacji ten sam warunek zapisujemy jako `{ $gt: [ "$wiek", 25 ] }` — różnica opisana szerzej na stronie Operatory wyrażeń, a pomylenie tych dwóch form to jeden z najczęstszych błędów początkujących.

Wykonanie `deleteMany({})` bez zamierzonego filtra. Pusty filtr pasuje do wszystkich dokumentów kolekcji — literówka albo pominięty warunek potrafi w ten sposób wyczyścić całą kolekcję jednym poleceniem.

Użycie `replaceOne` tam, gdzie chodziło o częściową aktualizację. `replaceOne` nadpisuje cały dokument — pola nieobecne w podanym zastępniku znikają, co bywa zaskoczeniem dla kogoś przyzwyczajonego do `$set`.

Powiązane: Architektura MongoDB: dokument, kolekcja, baza, replica set · Aggregation pipeline · Operatory wyrażeń · Indeksy w MongoDB

Źródła: dokumentacja MongoDB, rozdział „CRUD Operations" i „Query and Projection Operators"

Aggregation pipeline

Poziom: podstawy **Dotyczy: MongoDB 5.0–8.0** **Czas: ~25 min**

Potok agregacji to sposób, w jaki MongoDB przetwarza, grupuje i przekształca dane po stronie serwera, bez wyciągania ich najpierw do aplikacji.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
B2B	business to business: sprzedaż i systemy skierowane do innych firm, nie do klientów indywidualnych

Fiszka

Komendy wykonujemy w powłoce `mongosh`, po przełączeniu się na właściwą bazę.

TYLKO ODCZYT

Filtrowanie dokumentów: $match

```
db.zamowienia.aggregate([
  { $match: { kategoria: "elektronika", kwota: { $gt: 100 } } }
])
```

Przepuszczamy dalej wyłącznie dokumenty spełniające podany warunek. Składnia wewnątrz `$match` jest identyczna jak w zwykłym `find()`, opisanym na stronie Operacje na dokumentach, i obsługuje te same operatory zapytań.

Dlaczego `$match` dajemy jak najwcześniej. Umieszczenie `$match` na samym początku potoku ogranicza liczbę dokumentów, jakie muszą przejść przez dalsze, zwykle znacznie cięższe etapy takie jak `$group` czy `$lookup`. Dodatkowa korzyść to indeksy — potok korzysta z istniejącego indeksu kolekcji tylko dla `$match` i `$sort` umieszczonych na samym początku, zanim dane zostaną w jakikolwiek sposób przekształcone; `$match` dodany w środku potoku, po etapie który już zmienił kształt dokumentów, indeksu użyć nie może.

TYLKO ODCZYT

Wybór i przekształcanie pól: $project

```
db.zamowienia.aggregate([
  { $project: { _id: 0, klient: 1, kwota: 1 } }
])
```

Decydujemy, które pola znajdą się w dokumencie wyjściowym — wartość `1` oznacza pokazanie pola, `0` jego ukrycie. `$project` pozwala też tworzyć nowe pola wyliczane w miejscu:

```
db.zamowienia.aggregate([
  { $project: { klient: 1, kwotaBrutto: { $multiply: ["$kwota", 1.23] } } }
])
```

Pole `kwotaBrutto` powstaje tu jako iloczyn istniejącego pola `kwota` i stałej stawki — przykład wyrażenia obliczeniowego wewnątrz etapu, rozwiniętego szerzej na stronie Operatory wyrażeń.

TYLKO ODCZYT

Grupowanie i agregacja: $group

```
db.zamowienia.aggregate([
  { $group: {
      _id: "$klient",
      liczba: { $sum: 1 },
      sumaKwot: { $sum: "$kwota" },
      srednia: { $avg: "$kwota" },
      maks: { $max: "$kwota" }
  } }
])
```

Najważniejszy etap potoku. Pole `_id` określa klucz grupowania — tutaj grupujemy dokumenty per klient. Pozostałe pola to akumulatory liczone osobno w obrębie każdej grupy: `$sum: 1` zlicza liczbę dokumentów w grupie, `$sum: "$kwota"` sumuje wartości pola, a `$avg` i `$max` liczą odpowiednio średnią i wartość maksymalną. Ustawienie `_id: null` grupuje wszystkie dokumenty w jedną całość, dając agregację globalną zamiast rozbicia na grupy.

Przykład:

```
[
  { _id: 'Anna', liczba: 3, sumaKwot: 540, srednia: 180, maks: 300 },
  { _id: 'Jan', liczba: 1, sumaKwot: 120, srednia: 120, maks: 120 }
]
```

□□ TYLKO ODCZYT

Sortowanie wyniku: $sort

```
db.zamowienia.aggregate([
  { $group: { _id: "$klient", sumaKwot: { $sum: "$kwota" } } },
  { $sort: { sumaKwot: -1 } }
])
```

Porządkujemy dokumenty: `1` oznacza rosnąco, `-1` malejąco. Sortować można również po polach wyliczonych we wcześniejszych etapach potoku, tak jak tutaj po `sumaKwot` powstałym w poprzedzającym `$group`.

□□ TYLKO ODCZYT

Ograniczanie liczby wyników: $limit i $skip

```
db.zamowienia.aggregate([
  { $sort: { kwota: -1 } },
  { $skip: 10 },
  { $limit: 5 }
])
```

`$limit` zwraca co najwyżej podaną liczbę dokumentów, a `$skip` pomija pierwsze N z nich — w połączeniu z `$sort` służą do stronicowania wyników. `$limit` warto stawiać możliwie wcześnie w potoku, żeby odciążyć etapy, które są po nim.

□□ TYLKO ODCZYT

Zliczanie dokumentów: $count

```
db.zamowienia.aggregate([
  { $match: { status: "oplacone" } },
  { $count: "liczbaOplaconych" }
])
```

Zwracamy pojedynczy dokument z liczbą dokumentów, które dotarły do tego etapu potoku. Argument to nazwa pola, w jakim ma się znaleźć wynik.

Przykład:

```
[ { liczbaOplaconych: 214 } ]
```

□□ TYLKO ODCZYT

Rozwijanie tablic: $unwind

```
db.zamowienia.aggregate([
  { $unwind: "$produkty" }
])
```

Rozbijamy dokument zawierający tablicę na wiele osobnych dokumentów — po jednym na każdy element tej tablicy. Jeśli jedno zamówienie miało trzy produkty w polu `produkty`, po tym etapie powstają z niego trzy dokumenty, każdy z pojedynczym produktem zamiast całej tablicy. Etap niezbędny, gdy chcemy dalej grupować albo liczyć po elementach tablicy zamiast po całych dokumentach.

□□ TYLKO ODCZYT

Łączenie kolekcji: $lookup

```
db.zamowienia.aggregate([
  { $lookup: {
      from: "klienci",
      localField: "klient_id",
      foreignField: "_id",
      as: "daneKlienta"
  } }
])
```

Dołączamy dane z innej kolekcji — odpowiednik lewego złączenia (`LEFT JOIN`) znanego z baz relacyjnych i opisanego ogólnie na stronie LEFT JOIN i RIGHT JOIN. Dla każdego dokumentu zamówienia szukamy w kolekcji `klienci` dokumentów, w których `_id` odpowiada wartości pola `klient_id` z zamówienia. Dopasowania trafiają do nowego pola `daneKlienta` jako tablica, dlatego `$lookup` łączy się często z `$unwind` zaraz po nim, żeby tę tablicę spłaszczyć do pojedynczego obiektu.

TYLKO ODCZYT

Dodawanie pól wyliczanych: $addFields

```
db.zamowienia.aggregate([
  { $addFields: { kwotaBrutto: { $multiply: ["$kwota", 1.23] } } }
])
```

Dodajemy nowe pole do dokumentu, zachowując przy tym wszystkie dotychczasowe — w odróżnieniu od `$project`, gdzie pola do zachowania trzeba wymienić jawnie jedno po drugim. `$set` jest aliasem `$addFields` i działa z nim identycznie, bywa tylko uznawany za czytelniejszy w kontekście, gdzie potok już wcześniej używał `$match`/`$group`.

TYLKO ODCZYT

Wielokierunkowe podsumowanie w jednym przebiegu: $facet

```
db.zamowienia.aggregate([
  { $facet: {
      poStatusie: [ { $group: { _id: "$status", liczba: { $sum: 1 } } } ],
      top5: [ { $sort: { kwota: -1 } }, { $limit: 5 } ]
  } }
])
```

Uruchamiamy kilka niezależnych pod-potoków na tym samym zestawie dokumentów wejściowych i zwracamy ich wyniki razem, każdy pod osobnym kluczem — przydatne, gdy z jednego zapytania chcemy naraz dostać kilka różnych podsumowań bez wielokrotnego odpytywania kolekcji.

TYLKO ODCZYT

Zapisanie wyniku do kolekcji: $out i $merge

```
db.zamowienia.aggregate([
  { $group: { _id: "$klient", suma: { $sum: "$kwota" } } },
  { $out: "raport_klientow" }
])
```

Zapisujemy wynik potoku do osobnej kolekcji zamiast wyświetlać go na ekranie — przydatne przy budowaniu raportów cyklicznych. `$out` nadpisuje całą kolekcję docelową za każdym uruchomieniem; jeśli zamiast podmieniać całość chcemy aktualizować albo dopisywać dokumenty do istniejącej kolekcji, sięgamy po `$merge`, który wykonuje operację typu upsert zamiast pełnego nadpisania.

Z życia wzięte

□□ Z ŻYCIA WZIĘTE

Ola przygotowuje dla pana Zbigniewa comiesięczne zestawienie pięciu klientów B2B z największą sumą zamówień na mongo01. Pierwsza wersja potoku liczy sumy przez cała kolekcję zamówień w pętli po stronie aplikacji i trwa kilka minut, więc Ola przepisuje ją jako jeden potok agregacji z filtrowaniem daty na samym początku. Ten sam raport, który wcześniej blokował widok sklepu na czas liczenia, teraz wraca w niecałą sekundę, bo etapy $match i $sort na początku potoku korzystają z indeksu, zamiast przeglądać każdy dokument.

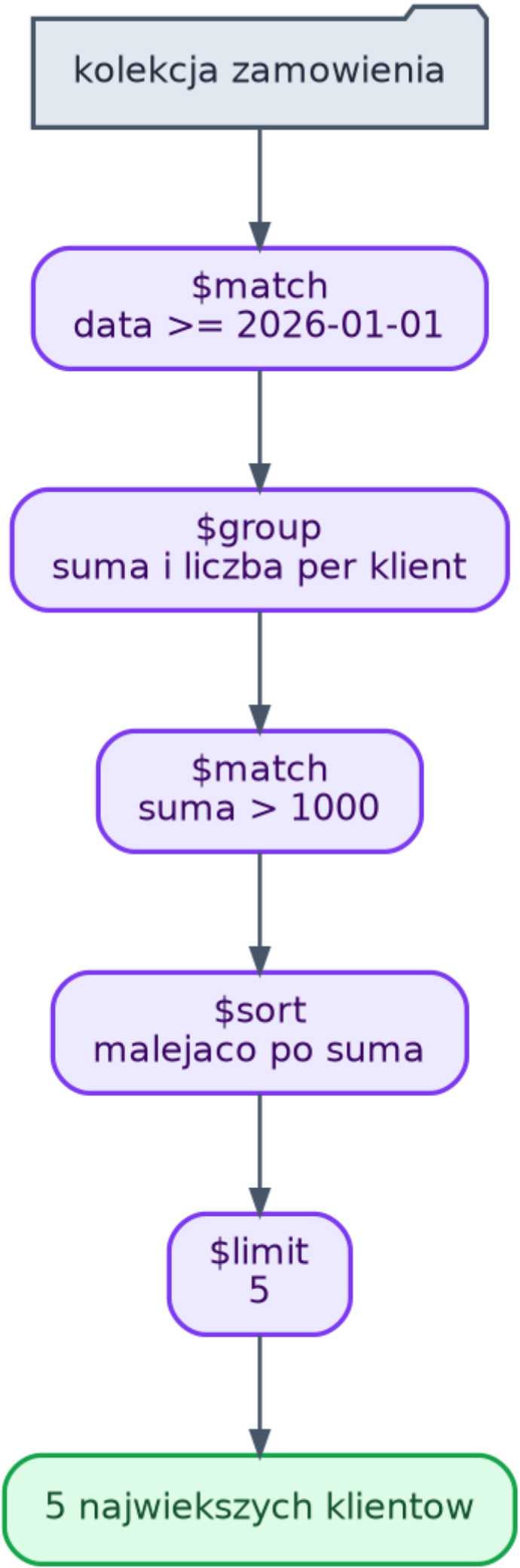
Po co to istnieje

Pojęcie: potok agregacji (aggregation pipeline). Mechanizm przetwarzania danych działający na zasadzie potoku znanego z powłoki Linux — tak jak `|` przekazuje wynik jednej komendy jako wejście następnej, tak `aggregate()` przepuszcza dokumenty przez kolejne etapy (stages), z których każdy przekształca dane i przekazuje wynik do następnego. Metoda `aggregate()` przyjmuje tablicę etapów, a kolejność tej tablicy ma bezpośrednie znaczenie — dokumenty płyną od pierwszego etapu do ostatniego, i etap piąty widzi już tylko to, co przepuściły i przekształciły etapy od pierwszego do czwartego.

```
db.zamowienia.aggregate([
  { $match: { status: "oplacone" } },
```

```
  { $group: { _id: "$klient", suma: { $sum: "$kwota" } } },
  { $sort: { suma: -1 } }
])
```

Powyższy przykład czyta się jak zdanie: wybieramy opłacone zamówienia, sumujemy kwoty per klient, sortujemy malejąco po sumie. Odwołania do pól dokumentu zapisujemy z prefiksem `$`, na przykład `"$kwota"` — to jest sposób, w jaki potok odróżnia nazwę pola dokumentu od zwykłego literału tekstowego.


kolekcja zamowienia
$match
data >= 2026-01-01
$group
suma i liczba per klient
$match
suma > 1000
$sort
malejaco po suma
$limit
5
5 najwiekszych klientow

 *Potok agregacji jako kolejne etapy: dokumenty płyną od kolekcji wejściowej przez match, group, match, sort i limit do wyniku (kliknięcie otwiera pełny rozmiar)*

Dobre praktyki i wydajność

Kilka zasad, które warto stosować przy budowaniu potoków, wykracza poza samo umieszczenie `$match` na początku. Przy dużych operacjach `$group` albo `$sort`, które przekraczają domyślny limit pamięci roboczej, dołączamy opcję pozwalającą korzystać z dysku:

```
db.zamowienia.aggregate([ /* etapy */ ], { allowDiskUse: true })
```

Plan wykonania potoku sprawdzamy tak samo jak plan zwykłego zapytania:

```
db.zamowienia.aggregate([ /* etapy */ ]).explain()
```

Przykład złożony — pełny potok

```
db.zamowienia.aggregate([
  { $match: { data: { $gte: ISODate("2026-01-01") } } },
  { $group: { _id: "$klient", suma: { $sum: "$kwota" }, liczba: { $sum: 1 } } },
  { $match: { suma: { $gt: 1000 } } },
  { $sort: { suma: -1 } },
  { $limit: 5 }
])
```

Potok czyta się jak zapytanie biznesowe: spośród zamówień od początku 2026 roku sumujemy kwoty i liczymy zamówienia per klient, zostawiamy tylko klientów z sumą powyżej 1000, sortujemy malejąco i pokazujemy pięciu największych. Drugi `$match` w tym potoku filtruje już po polu `suma` wyliczonym w `$group` — czego zwykły `find()` zrobić nie potrafi, bo pole `suma` nie istnieje w żadnym dokumencie wejściowym, powstaje dopiero w trakcie potoku.

Częste błędy

Umieszczenie `$match` po ciężkich etapach zamiast przed nimi. Filtr działający na końcu potoku wciąż zmusza wcześniejsze etapy do przetworzenia wszystkich dokumentów, które i tak zostaną odrzucone — koszt pozostaje ten sam, niezależnie od tego, że wynik końcowy jest identyczny.

Oczekiwanie, że `$match` po `$group` skorzysta z indeksu. Indeks działa tylko dla `$match`/`$sort` umieszczonych zanim jakikolwiek etap przekształci dokumenty — drugi `$match` w przykładzie złożonym wyżej, filtrujący po `suma`, zawsze wykona pełne przejście po już zgrupowanych danych.

Nieoczekiwane nadpisanie kolekcji przez `$out`. `$out` zastępuje całą zawartość kolekcji docelowej, nie dopisuje do niej — pomyłka co do tego zachowania potrafi bezpowrotnie skasować dane, które ktoś inny trzymał pod tą samą nazwą.

Powiązane: Operacje na dokumentach · Operatory wyrażeń · Architektura MongoDB: dokument, kolekcja, baza, replica set · Indeksy w MongoDB

Źródła: notatki własne administratora MongoDB · dokumentacja MongoDB, rozdział „Aggregation Pipeline"

Operatory wyrażeń (logika i formatowanie wewnątrz etapów)

Poziom: podstawy **Dotyczy: MongoDB 5.0–8.0** **Czas: ~20 min**

Operatory, którymi wewnątrz etapów potoku agregacji liczymy wartości — warunki, działania arytmetyczne, formatowanie tekstu i dat — zamiast tylko filtrować dokumenty.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
B2B	business to business: sprzedaż i systemy skierowane do innych firm, nie do klientów indywidualnych
CSV	w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami
UTC	Coordinated Universal Time: uniwersalny czas bez strefy, punkt odniesienia dla wszystkich stref czasowych na świecie

Fiszka

Wszystkie poniższe operatory wykonują się w powłoce `mongosh`, w ramach etapu potoku agregacji.

TYLKO ODCZYT

Warunek if/then/else: $cond

```
db.zamowienia.aggregate([
  { $addFields: {
      kategoriaKwoty: {
        $cond: { if: { $gte: ["$kwota", 500] }, then: "duze", else: "male" }
```

```
      }
  } }
])
```

Zwracamy jedną z dwóch wartości w zależności od spełnienia warunku — odpowiednik `if`/`else`. Powyżej każdemu zamówieniu dopisywane jest pole `kategoriaKwoty`, równe `"duze"`, gdy kwota wynosi co najmniej 500, w przeciwnym razie `"male"`.

Skrócona składnia tablicowa, gdzie kolejność argumentów to warunek, wartość przy prawdzie i wartość przy fałszu:

```
{ $cond: [ { $gte: ["$kwota", 500] }, "duze", "male" ] }
```

Przykład:

```
[
  { _id: ObjectId('...'), kwota: 620, kategoriaKwoty: 'duze' },
  { _id: ObjectId('...'), kwota: 340, kategoriaKwoty: 'male' }
]
```

TYLKO ODCZYT

Wielokrotny wybór: $switch

```
db.zamowienia.aggregate([
  { $addFields: {
      prog: {
        $switch: {
          branches: [
            { case: { $gte: ["$kwota", 1000] }, then: "premium" },
            { case: { $gte: ["$kwota", 500] }, then: "standard" }
          ],
          default: "podstawowy"
        }
      }
  } }
])
```

Sprawdzamy kolejno warunki z listy `branches` i zwracamy wynik pierwszego pasującego. Jeśli żaden warunek nie pasuje, zwracamy wartość z `default`. To czytelniejsza alternatywa dla

kilku zagnieżdżonych `$cond` naraz. Warunki są sprawdzane od góry listy w dół, więc ich kolejność ma znaczenie — dokument o kwocie 1200 pasujący jednocześnie do obu warunków dostanie wynik pierwszego z nich, `"premium"`.

TYLKO ODCZYT

Wartość domyślna dla braków: $ifNull

```
db.zamowienia.aggregate([
  { $addFields: { rabat: { $ifNull: ["$rabat", 0] } } }
])
```

Zwracamy pierwszą wartość, jeśli nie jest ona `null` ani nieobecna w dokumencie; w przeciwnym razie zwracamy wartość zastępczą podaną jako drugi argument. Tutaj brakujące pole `rabat` zostaje uzupełnione zerem, co zabezpiecza dalsze obliczenia arytmetyczne przed błędami wynikającymi z wartości `null`.

TYLKO ODCZYT

Akumulatory grupujące: $sum, $avg, $min, $max

```
{ $group: {
    _id: "$klient",
    liczba: { $sum: 1 },
    sredniaKwota: { $avg: "$kwota" },
    minimalna: { $min: "$kwota" },
    maksymalna: { $max: "$kwota" }
} }
```

Cztery podstawowe akumulatory liczone w obrębie każdej grupy etapu `$group`, opisanego na stronie Aggregation pipeline. `$sum: 1` zlicza liczbę dokumentów w grupie, `$sum` z nazwą pola sumuje jego wartości, a `$avg`, `$min`, `$max` liczą odpowiednio średnią i wartości skrajne.

TYLKO ODCZYT

Pierwszy i ostatni dokument grupy: $first i $last

```
{ $group: { _id: "$klient", pierwszeZamowienie: { $first: "$data" }, ostatnieZamowienie: { $last: "$data" } } }
```

Zwracają wartość pola z pierwszego albo ostatniego dokumentu napotkanego w obrębie grupy — mają sens tylko wtedy, gdy dokumenty trafiają do `$group` w ustalonej kolejności, czyli po wcześniejszym `$sort` w tym samym potoku.

TYLKO ODCZYT

Zbieranie wartości do tablicy: $push i $addToSet

```
{ $group: { _id: "$klient", produkty: { $push: "$produkt" } } }
```

Zbieramy wartości pola ze wszystkich dokumentów grupy do jednej tablicy, z zachowaniem duplikatów i kolejności napotkania.

```
{ $group: { _id: "$klient", produktyUnikalne: { $addToSet: "$produkt" } } }
```

Działa identycznie jak `$push`, ale pomija duplikaty — wynikowa tablica zawiera każdą wartość tylko raz, w kolejności nieokreślonej.

TYLKO ODCZYT

Sklejanie tekstu: $concat

```
{ $concat: ["$imie", " ", "$nazwisko"] }
```

Sklejamy kilka fragmentów tekstu w jeden ciąg. **Uważaj:** jeśli którykolwiek z fragmentów ma wartość `null`, cały wynik `$concat` również będzie `null` — warto opakować niepewne pole w `$ifNull` przed sklejeniem.

TYLKO ODCZYT

Wycinek tekstu: $substr

```
{ $substr: ["$kod", 0, 3] }
```

Zwracamy fragment tekstu — pierwszy argument to pole źródłowe, drugi pozycja startowa liczona od zera, trzeci długość wycinka.

TYLKO ODCZYT

Zmiana wielkości liter: $toUpper

```
{ $toUpper: "$status" }
```

Zamieniamy wartość pola tekstowego na same wielkie litery — odpowiednik jest też `$toLower`, działający analogicznie w drugą stronę.

TYLKO ODCZYT

Formatowanie daty jako tekstu: $dateToString

```
db.zamowienia.aggregate([
  { $addFields: {
      dataTekst: {
        $dateToString: { format: "%Y-%m-%d", date: "$data", timezone: "Europe/Warsaw" }
      }
  } }
])
```

Zamieniamy pole typu data na sformatowany tekst według wzorca podanego w `format`. Opcjonalny `timezone` przelicza datę na wskazaną strefę czasową przed sformatowaniem — bez niego daty są traktowane jako UTC.

Przykład:

```
[ { _id: ObjectId('...'), data: ISODate('2026-09-15T22:30:00Z'), dataTekst: '2026-09-16'
} ]
```

UTC = Coordinated Universal Time: uniwersalny czas bez strefy, punkt odniesienia dla wszystkich stref czasowych na świecie

TYLKO ODCZYT

Różnica między dwiema datami: $dateDiff

```
{ $dateDiff: { startDate: "$data", endDate: "$$NOW", unit: "day" } }
```

Liczymy odstęp między dwiema datami w zadanej jednostce (`day`, `hour`, `month` i inne). Zmienna `$$NOW` oznacza bieżący czas serwera w chwili wykonania potoku — podwójny prefiks

`$$` odróżnia zmienną systemową albo lokalną od zwykłego odwołania do pola dokumentu.

□□ TYLKO ODCZYT

Konwersja typów: $convert i $toInt

```
{ $toInt: "$kwotaTekst" }
```

Zamieniamy tekst na liczbę całkowitą — niezbędne przy porządkowaniu danych zaimportowanych jako tekst, na przykład z pliku CSV przez `mongoimport`, opisany na stronie Backup i odtwarzanie: mongodump i mongorestore.

Bardziej elastyczny jest `$convert`, pozwalający wskazać typ docelowy i wartość zastępczą na wypadek błędu konwersji:

```
{ $convert: { input: "$kwotaTekst", to: "double", onError: 0, onNull: 0 } }
```

Bez `onError` błąd konwersji (na przykład tekst niebędący liczbą) przerywa cały potok wyjątkiem — `onError` zamienia go na spokojnie zwróconą wartość zastępczą.

□□ **CSV** = w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami

□□ TYLKO ODCZYT

Dostęp do elementu tablicy po indeksie: $arrayElemAt

```
{ $arrayElemAt: ["$produkty", 0] }
```

Zwracamy element tablicy o wskazanym indeksie, liczonym od zera. Indeks ujemny liczy od końca tablicy — `-1` zwraca ostatni element.

□□ TYLKO ODCZYT

Liczba elementów tablicy: $size

```
{ $size: "$produkty" }
```

Zwracamy liczbę elementów w tablicy — bywa przydatne po etapie `$lookup`, żeby policzyć liczbę dopasowań znalezionych w innej kolekcji bez rozwijania ich przez `$unwind`.

TYLKO ODCZYT

Filtrowanie elementów tablicy: $filter

```
{ $filter: { input: "$produkty", as: "p", cond: { $gt: ["$$p.cena", 100] } } }
```

Zwracamy podzbiór tablicy spełniający podany warunek. Zmienna pętli `$$p` (nazwa dowolna, zadeklarowana w `as`) ma podwójny prefiks `$$`, bo odnosi się do zmiennej lokalnej wprowadzonej przez sam operator, a nie do pola całego dokumentu.

TYLKO ODCZYT

Przekształcanie elementów tablicy: $map

```
{ $map: { input: "$produkty", as: "p", in: "$$p.nazwa" } }
```

Przekształcamy każdy element tablicy według wyrażenia podanego w `in` — tutaj wyciągamy samą nazwę z każdego obiektu produktu, zamieniając tablicę obiektów na tablicę samych nazw.

Z życia wzięte

Z ŻYCIA WZIĘTE

Marek prosi o raport, w którym każdy produkt w katalogu B2B ma pokazaną cenę z rabatem naliczonym warunkowo, zależnie od kategorii i ilości na stanie. Ola buduje to jednym etapem $project z zagnieżdżonym $cond zamiast liczyć rabaty po stronie aplikacji dla każdego z kilkudziesięciu tysięcy produktów osobno. Zapytanie, które w pierwszej wersji trwało kilkanaście sekund, po przeniesieniu logiki do operatora wewnątrz potoku wraca niemal natychmiast.

Po co to istnieje

Pojęcie: operator wyrażenia. Operatory wyrażeń to konstrukcje używane wewnątrz etapów potoku agregacji, opisanego na stronie Aggregation pipeline — najczęściej `$project`, `$addFields` i `$group` — do obliczania nowych wartości: warunków, działań arytmetycznych,

> formatowania tekstu i dat. Różnią się od operatorów zapytań używanych w `find()` i `$match`, które jedynie filtrują dokumenty, nie tworząc niczego nowego. Operatory wyrażeń zawsze coś obliczają i zwracają.

Kluczowa różnica składni: operatory porównań w wyrażeniach przyjmują argumenty w postaci tablicy, na przykład `{ $gt: ["$kwota", 100] }`. To nie to samo co forma zapytania `{ kwota: { $gt: 100 } }` używana w `$match` i `find()`, opisana na stronie Operacje na dokumentach. Mylenie tych dwóch form jest najczęstszym błędem, jaki popełnia się przy pierwszym kontakcie z potokiem agregacji.

Przykład złożony — operatory w akcji

```
db.zamowienia.aggregate([
  { $addFields: {
      brutto: { $round: [ { $multiply: ["$kwota", 1.23] }, 2 ] },
      miesiac: { $dateToString: { format: "%Y-%m", date: "$data" } },
      etykieta: { $cond: [ { $gte: ["$kwota", 500] }, "VIP", "zwykly" ] },
      rabat: { $ifNull: ["$rabat", 0] }
  } }
])
```

Jeden etap `$addFields` liczy tu kwotę brutto zaokrągloną do dwóch miejsc po przecinku, wyciąga miesiąc zamówienia jako tekst, nadaje etykietę na podstawie progu kwoty i uzupełnia brakujący rabat zerem. Pokazuje to, jak operatory wyrażeń składa się ze sobą, żeby w jednym przebiegu przygotować dane gotowe do dalszego raportu.

Częste błędy

Użycie formy zapytania zamiast formy wyrażenia. `{ kwota: { $gt: 100 } }` wewnątrz `$addFields` czy `$project` nie zadziała tak jak w `$match` — trzeba użyć `{ $gt: ["$kwota", 100] }`, z argumentami w tablicy.

Pojedynczy `$` zamiast `$$` przy zmiennej lokalnej. Wewnątrz `$filter` czy `$map` odwołanie do zmiennej pętli z pojedynczym `$` (`$p.cena` zamiast `$$p.cena`) MongoDB potraktuje jako próbę odwołania do pola dokumentu o nazwie `p`, którego nie ma, i zwróci `null` zamiast oczekiwanej wartości.

Konwersja bez obsługi błędu. Zwykłe `$toInt` na polu, które czasem zawiera tekst nienadający się do konwersji, przerywa cały potok wyjątkiem — w danych o niepewnej jakości bezpieczniejszy jest `$convert` z ustawionym `onError`.

Powiązane: Aggregation pipeline · Operacje na dokumentach

Źródła: notatki własne administratora MongoDB · dokumentacja MongoDB, rozdział „Aggregation Pipeline Operators"

Indeksy w MongoDB

Poziom: podstawy **Dotyczy: MongoDB 5.0–8.0** **Czas: ~20 min**

Czym jest indeks w bazie dokumentowej i dlaczego, tak samo jak w bazie relacyjnej opisanej na stronie Czym jest indeks — i dlaczego to nie jest „przyspieszacz tabeli", nie jest to uniwersalny przycisk przyspieszający każde zapytanie.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
B2B	business to business: sprzedaż i systemy skierowane do innych firm, nie do klientów indywidualnych
COLLSCAN	collection scan: w MongoDB przejrzenie każdego dokumentu kolekcji po kolei, bez pomocy indeksu
IXSCAN	index scan: w MongoDB skorzystanie z istniejącego indeksu, żeby przejść od razu do pasujących dokumentów zamiast przeglądać wszystkie

Fiszka

Komendy wykonujemy w powłoce `mongosh`, po przełączeniu się na właściwą bazę.

TYLKO ODCZYT

Utworzenie indeksu na pojedynczym polu: createIndex

```
db.uzytkownicy.createIndex({ email: 1 })
```

Zakładamy indeks na wskazanym polu — wartość `1` oznacza porządek rosnący, `-1` malejący. Dla pojedynczego indeksu kierunek nie ma znaczenia dla szybkości wyszukiwania po nim, ma znaczenie dopiero przy sortowaniu i przy indeksach złożonych.

Przykład:

```
'email_1'
```

Zwrócony tekst to nazwa nowo utworzonego indeksu, nadana automatycznie ze wzorca `nazwaPola_kierunek`.

🔒 TYLKO ODCZYT

Utworzenie indeksu złożonego: kilka pól naraz

```
db.zamowienia.createIndex({ klient_id: 1, data: -1 })
```

> **Pojęcie: indeks złożony.** Indeks obejmujący więcej niż jedno pole naraz, w ustalonej kolejności pól. Kolejność ta rządzi się zasadą prefiksu: indeks złożony `{ klient_id: 1, data: -1 }` przyspiesza zapytania filtrujące po samym `klient_id`, jak i po `klient_id` razem z `data`, ale nie przyspiesza zapytania filtrującego wyłącznie po `data` — tak jak nie da się skorzystać ze spisu treści książki posortowanego według rozdziałów, żeby szybko znaleźć stronę po samym numerze akapitu. Kolejność pól przy zakładaniu indeksu warto więc dobierać zgodnie z tym, jak faktycznie wygląda najczęstszy filtr w zapytaniach aplikacji.

🔒 TYLKO ODCZYT

Utworzenie indeksu z wymuszeniem unikalności: unique

```
db.uzytkownicy.createIndex({ email: 1 }, { unique: true })
```

Wymuszamy, żeby żadne dwa dokumenty w kolekcji nie miały tej samej wartości wskazanego pola — próba wstawienia albo aktualizacji dokumentu, która złamałaby tę unikalność, kończy się błędem zamiast cichego zapisania duplikatu.

Przykład błędu:

```
MongoServerError: E11000 duplicate key error collection: appdb.uzytkownicy
index: email_1 dup key: { email: "anna@zielona.example" }
```

🔒 TYLKO ODCZYT

Utworzenie indeksu częściowego: partialFilterExpression

```
db.zamowienia.createIndex(
  { status: 1 },
  { partialFilterExpression: { status: "oplacone" } }
)
```

> **Pojęcie: indeks częściowy.** Indeks obejmujący tylko podzbiór dokumentów kolekcji, spełniających podany warunek — tutaj wyłącznie dokumenty o statusie `"oplacone"`. Mniejszy indeks zajmuje mniej miejsca i jest tańszy w utrzymaniu przy zapisach, kosztem tego, że przyspiesza wyłącznie zapytania, których filtr logicznie zawiera się w warunku indeksu.

□□ TYLKO ODCZYT

Utworzenie indeksu z automatycznym wygasaniem: TTL

```
db.sesje.createIndex({ utworzono: 1 }, { expireAfterSeconds: 3600 })
```

> **Pojęcie: indeks TTL.** Indeks, który poza przyspieszaniem wyszukiwania dodatkowo powoduje automatyczne usunięcie dokumentu, gdy od wartości daty we wskazanym polu upłynie podana liczba sekund. Wymaga, żeby indeksowane pole faktycznie zawierało wartość typu data. Sięgamy po to przy danych z naturalnym terminem ważności — sesjach logowania, tokenach, logach zdarzeń — zamiast pisać osobne zadanie cykliczne czyszczące starą zawartość.

□□ **TTL** = Time To Live: czas życia rekordu lub danych

□□ TYLKO ODCZYT

Utworzenie indeksu tekstowego: text

```
db.artykuly.createIndex({ tresc: "text" })
```

> **Pojęcie: indeks tekstowy.** Indeks przystosowany do wyszukiwania pełnotekstowego zamiast dokładnego dopasowania wartości — obsługuje wyszukiwanie słów kluczowych z pominięciem odmiany słów podstawowych i z ignorowaniem wyrazów bardzo częstych (tak zwanych stop words). Zapytanie korzystające z takiego indeksu używa operatora `$text`:

```
db.artykuly.find({ $text: { $search: "wydajność indeksów" } })
```

🔒 TYLKO ODCZYT

Utworzenie indeksu geoprzestrzennego: 2dsphere

```
db.lokalizacje.createIndex({ wspolrzedne: "2dsphere" })
```

> **Pojęcie: indeks geoprzestrzenny.** Indeks przystosowany do zapytań o położenie geograficzne — najbliższe punkty względem podanych współrzędnych, punkty leżące wewnątrz wskazanego obszaru. Pole indeksowane musi być zapisane w formacie GeoJSON.

🔒 TYLKO ODCZYT

Wylistowanie indeksów kolekcji: getIndexes

```
db.uzytkownicy.getIndexes()
```

Wyświetlamy wszystkie indeksy założone na kolekcji, razem z ich pełną definicją i opcjami.

Przykład:

```
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { email: 1 }, name: 'email_1', unique: true }
]
```

Indeks `_id_` na polu `_id` istnieje zawsze, zakładany automatycznie w chwili utworzenia kolekcji, i nie da się go usunąć.

🔒 TYLKO ODCZYT

Usunięcie indeksu: dropIndex

```
db.uzytkownicy.dropIndex("email_1")
```

Usuwamy indeks po jego nazwie, odczytanej wcześniej z `getIndexes()`. Usunięcie indeksu jest natychmiastowe i nieodwracalne — odtworzenie go wymaga ponownego przejścia przez całą kolekcję.

TYLKO ODCZYT

Sprawdzenie planu wykonania zapytania: explain

```
db.zamowienia.find({ status: "oplacone" }).explain("executionStats")
```

Zwracamy szczegółowy plan tego, jak silnik faktycznie wykonał (albo wykonałby) zapytanie — czy skorzystał z indeksu, ile dokumentów faktycznie przejrzał i ile z nich zwrócił jako pasujące.

Przykład dla zapytania bez indeksu:

```
{
  executionStats: {
    executionStages: {
      stage: 'COLLSCAN',
      nReturned: 214,
      docsExamined: 98211
    }
  }
}
```

Ten sam filtr po założeniu indeksu na `status`:

```
{
  executionStats: {
    executionStages: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        indexName: 'status_1',
        nReturned: 214
      }
    },
    nReturned: 214,
```

```
    docsExamined: 214
  }
}
```

Z życia wzięte

Z ŻYCIA WZIĘTE

W środę po południu wyszukiwarka produktów w sklepie B2B zaczyna odpowiadać zauważalnie wolniej, chociaż liczba produktów w katalogu na mongo01 nie zmieniła się drastycznie od miesiąca. Tomek sprawdza explain() zapytania wyszukującego po nazwie i kategorii jednocześnie i widzi COLLSCAN zamiast IXSCAN, bo istniejący indeks obejmuje tylko pole nazwa, nie parę pól używaną teraz przez frontend. Zakłada indeks złożony na obu polach naraz i czas odpowiedzi wyszukiwarki wraca do wcześniejszego poziomu.

Jak czytać COLLSCAN i IXSCAN

Pojęcie: COLLSCAN. Skrót od collection scan — silnik przegląda każdy dokument kolekcji po kolei, sprawdzając, czy pasuje do filtra. Widoczna w przykładzie wyżej liczba `docsExamined: 98211` przy `nReturned: 214` pokazuje koszt takiego podejścia: silnik musiał przejrzeć niemal sto tysięcy dokumentów, żeby znaleźć dwieście czternaście pasujących.

Pojęcie: IXSCAN. Skrót od index scan — silnik korzysta z istniejącego indeksu, żeby od razu przejść do dokumentów pasujących do filtra, zamiast przeglądać wszystkie. W drugim przykładzie `docsExamined` spada do dokładnie tylu, ile faktycznie zwrócono — indeks doprowadził silnik prosto do właściwych dokumentów.

Różnica między tymi dwoma planami na dużej kolekcji bywa rzędu wielkości albo więcej — `COLLSCAN` na kolekcji z milionami dokumentów przy filtrze zwracającym garstkę z nich jest pierwszym sygnałem, że brakuje tam odpowiedniego indeksu.

Częste błędy

Zakładanie, że każdy indeks przyspiesza każde zapytanie. Indeks przyspiesza konkretny wzorzec filtrowania i sortowania, zgodny z kolejnością jego pól — zapytanie filtrujące po polu spoza tej kolejności prefiksu wciąż skończy jako `COLLSCAN`, mimo że jakiś indeks na kolekcji istnieje.

Zbyt wiele indeksów na kolekcji o dużym ruchu zapisu. Każdy indeks trzeba zaktualizować przy każdym zapisie do kolekcji — kolekcja z kilkunastoma indeksami płaci za to zauważalnym spowolnieniem operacji `insert`/`update`, nawet jeśli same odczyty stają się szybsze.

Odczyt `explain()` bez trybu `executionStats`. Domyślny tryb `explain()` (`queryPlanner`) pokazuje tylko wybrany plan, bez rzeczywistych liczb `docsExamined`/`nReturned` — bez trybu `executionStats` trudno ocenić, czy wybrany plan faktycznie jest efektywny w praktyce, a nie tylko teoretycznie poprawny.

Powiązane: Czym jest indeks — i dlaczego to nie jest „przyspieszacz tabeli" · Operacje na dokumentach · Aggregation pipeline

Źródła: dokumentacja MongoDB, rozdział „Indexes" i „Query Plans"