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.

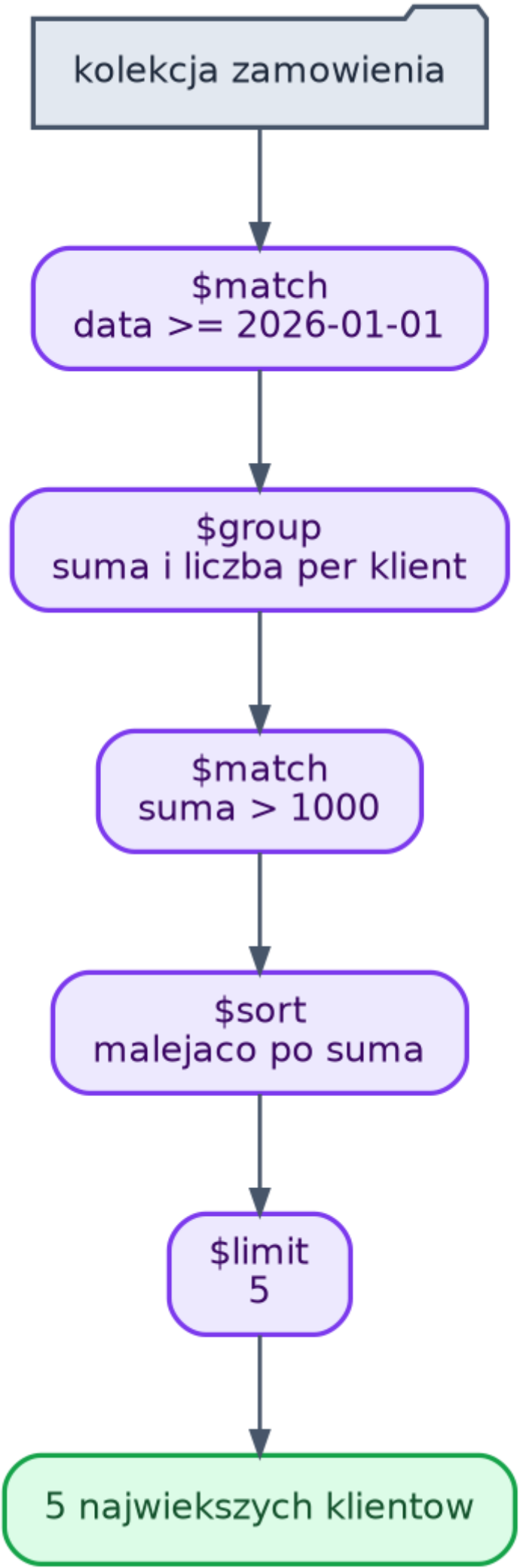
Po co to istnieje

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"

Revision #42
Created 2026-06-09 09:32:24 UTC by Tomasz Konieczny
Updated 2026-10-26 14:09:59 UTC by Tomasz Konieczny