Aggregation pipeline

Wprowadzenie:

> Aggregation pipeline to mechanizm przetwarzania danych w MongoDB, działający na zasadzie potoku (jak `|` w powłoce Linux). Dokumenty przechodzą przez kolejne etapy (*stages*), z których każdy przekształca dane i przekazuje wynik do następnego. Pozwala to grupować, filtrować, łączyć i przeliczać dane po stronie serwera, bez wyciągania ich do aplikacji.

Wszystkie poniższe komendy wykonuje się **wewnątrz powłoki mongosh**, więc wyglądają tak samo niezależnie od tego, czy serwer działa baremetal, czy w kontenerze (sposób wejścia do powłoki opisano w instrukcji „Podstawowe komendy mongosh", sekcja 1).

1. Struktura potoku – `aggregate()`

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

Metoda `aggregate()` przyjmuje tablicę etapów. Kolejność ma znaczenie — dokumenty płyną od pierwszego etapu do ostatniego. Powyższy przykład czyta się jak zdanie: „wybierz opłacone zamówienia, zsumuj kwoty per klient, posortuj malejąco po sumie". Odwołania do pól dokumentu zapisuje się z prefiksem `$`, np. `"$kwota"`.

2. Etap `$match` – filtrowanie

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

Przepuszcza dalej tylko dokumenty spełniające warunek. Składnia jest identyczna jak w zwykłym `find()` i obsługuje te same operatory (`$gt`, `$in` itd.). Najlepiej umieszczać `$match` na początku potoku — ogranicza liczbę dokumentów przetwarzanych przez dalsze, cięższe etapy.

3. Etap `$project` – wybór i przekształcanie pól

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

Decyduje, które pola znajdą się w wyniku (`1` = pokaż, `0` = ukryj). Pozwala też tworzyć pola wyliczane:

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

Tutaj powstaje nowe pole `kwotaBrutto` jako iloczyn istniejącej kwoty i stawki — przykład wyrażenia obliczeniowego wewnątrz potoku.

4. Etap `$group` – grupowanie i agregacja

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

Najważniejszy etap. Pole `_id` określa klucz grupowania (tutaj: per klient). Pozostałe pola to *akumulatory* liczone w obrębie grupy: `$sum: 1` zlicza dokumenty, `$sum: "$kwota"` sumuje kwoty, a `$avg`, `$min`, `$max` liczą odpowiednio średnią i skrajne wartości. Ustawienie `_id: null` grupuje wszystkie dokumenty w jedną całość (agregacja globalna).

Przykład wyniku:

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

> **Akumulatory do zapamiętania:** `$sum`, `$avg`, `$min`, `$max`, `$first`, `$last`, `$push` (zbiera wartości do tablicy) oraz `$addToSet` (jak `$push`, ale bez duplikatów).

> **Screen:** wynik grupowania per klient.

5. Etap `$sort` – sortowanie

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

Porządkuje dokumenty: `1` = rosnąco, `-1` = malejąco. Można sortować również po polach wyliczonych we wcześniejszych etapach (jak tutaj po `sumaKwot`).

6. Etapy `$limit` i `$skip` – ograniczanie wyników

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

```
])
```

`$limit` zwraca co najwyżej podaną liczbę dokumentów, a `$skip` pomija pierwsze N. W połączeniu z `$sort` służą do stronicowania wyników. `$limit` warto stawiać możliwie wcześnie, by odciążyć dalsze etapy.

7. Etap `$count` – zliczanie dokumentów

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

Zwraca pojedynczy dokument z liczbą dokumentów, które dotarły do tego etapu. Argument to nazwa pola, w którym znajdzie się wynik.

8. Etap `$unwind` – rozwijanie tablic

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

Rozbija dokument zawierający tablicę na wiele dokumentów — po jednym na każdy element tablicy. Jeśli zamówienie miało trzy produkty w polu `produkty`, powstaną trzy dokumenty, każdy z pojedynczym produktem. Niezbędne, gdy chcemy grupować lub liczyć po elementach tablicy.

9. Etap `$lookup` – łączenie kolekcji (JOIN)

```
db.zamowienia.aggregate([
  { $lookup: {
      from: "klienci",
```

```
    localField: "klient_id",
    foreignField: "_id",
    as: "dane_klienta"
  } }
])
```

Dołącza dane z innej kolekcji — odpowiednik lewego złączenia (LEFT JOIN). Dla każdego zamówienia szuka w kolekcji `klienci` dokumentów, w których `_id` równa się polu `klient_id` z zamówienia. Dopasowania trafiają do nowego pola `dane_klienta` jako **tablica** — często łączy się `$lookup` z `$unwind`, by ją spłaszczyć.

10. Etapy `$addFields` / `$set` – dodawanie pól wyliczanych

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

Dodaje nowe pole do dokumentu, zachowując wszystkie dotychczasowe (w przeciwieństwie do `$project`, gdzie pola trzeba wymienić jawnie). `$set` to alias `$addFields` — działa identycznie i bywa czytelniejszy.

11. Etapy `$out` / `$merge` – zapis wyników

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

Zapisuje wynik potoku do osobnej kolekcji zamiast wyświetlać go na ekranie — przydatne do budowania raportów cyklicznych.

>

> **Uwaga:** `$out` **nadpisuje** całą kolekcję docelową. Jeśli chcesz aktualizować/dołączać dokumenty zamiast podmieniać całość, użyj `$merge`, który wykonuje upsert na istniejącej kolekcji.

12. 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 zsumuj kwoty i policz zamówienia per klient, zostaw tylko klientów z sumą powyżej 1000, posortuj malejąco i pokaż pięciu największych". Zwróć uwagę na drugi `$match` — filtruje już po polu `suma` wyliczonym w `$group`, czego zwykły `find()` nie potrafi.

> **Screen:** wynik pełnego potoku (top 5 klientów).

13. Dobre praktyki i wydajność

Kilka zasad, które warto stosować przy budowaniu potoków:

- **Filtruj wcześnie** – `$match` i `$limit` na początku ograniczają liczbę dokumentów przekazywanych do cięższych etapów (`$group`, `$lookup`).
- **Indeksy** – potok korzysta z indeksów tylko dla `$match` i `$sort` umieszczonych na *początku*, zanim dane zostaną przekształcone.
- **Analiza planu** – sprawdź, jak MongoDB wykona potok:

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

- **Duże operacje** – przy obszernych `$group`/`$sort` przekraczających limit pamięci dołącz opcję pozwalającą korzystać z dysku:

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

Podsumowanie praktyczne

```
// Szkielet potoku: filtruj → grupuj → filtruj po agregacie → sortuj → ogranicz
db.zamowienia.aggregate([
  { $match: { status: "oplacone" } },
  { $group: { _id: "$klient", suma: { $sum: "$kwota" } } },
  { $match: { suma: { $gt: 1000 } } },
  { $sort: { suma: -1 } },
  { $limit: 10 }
])
```

Revision #1
Created 9 June 2026 09:32:24 by Tomasz Konieczny
Updated 9 June 2026 09:33:01 by Tomasz Konieczny