

Sprawozdanie – Etap III

Wykorzystanie bazy dokumentowej MongoDB

Wybrany zbiór danych

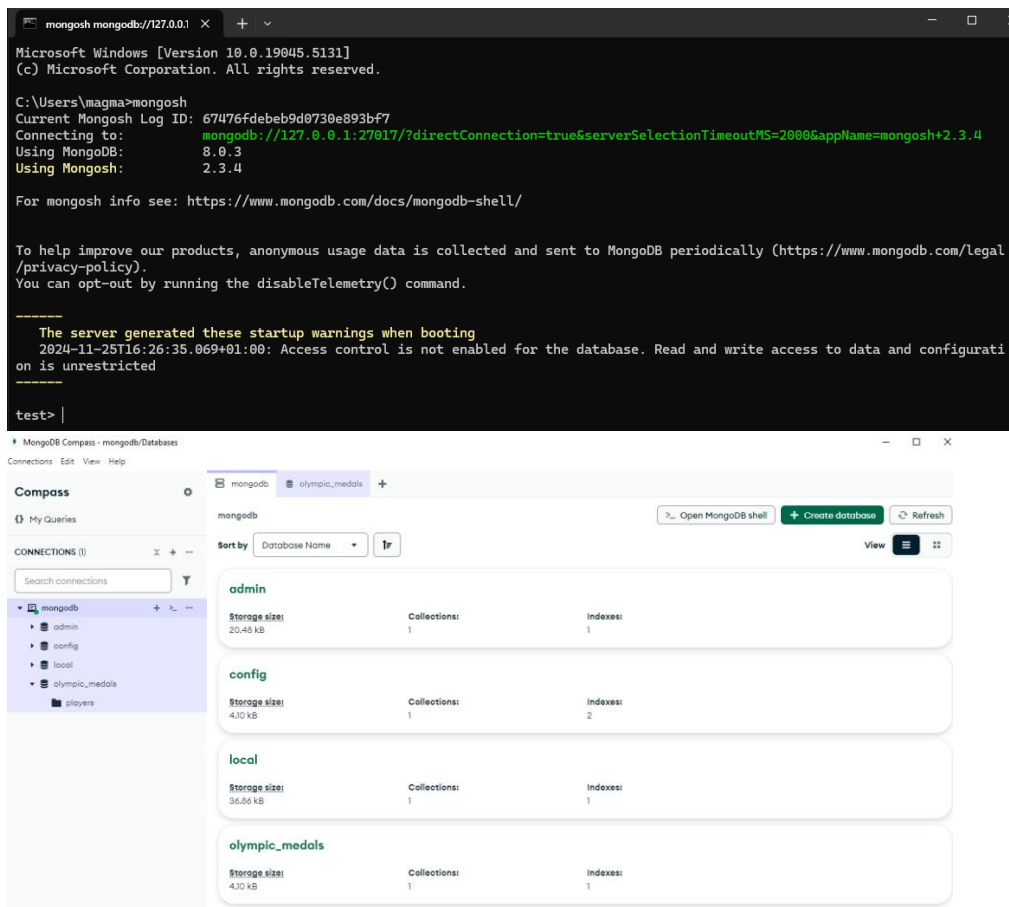
Wybrano zbiór danych dotyczący medali w Letnich Igrzyskach Olimpijskich w latach 1976-2008.

Link do zbioru: <https://www.kaggle.com/datasets/divyansh22/summer-olympics-medals/data>

Wybrano ten zbiór, ponieważ był odpowiednich rozmiarów (posiada np. 11338 unikalnych zawodników), posiadał niemałą ilość pól (10 pól) i nie posiadał błędów ani nieprawidłowości. Ponadto, pola bazy wydawały się odpowiednie do projektu – np. pola Sport i Event wykorzystano później w zagnieżdżonych dokumentach.

Instalacja MongoDB

Zainstalowano MongoDB w wersji 8.0.3 (najnowsza dostępna na stronie) w wersji Community. Wybrano pełną opcję instalacji, więc zainstalowany został między innymi również MongoDB Compass (GUI do operacji CRUD). Ponadto, doinstalowano mongosh, czyli CLI dla MongoDB.



(Kolekcję players zmieniono na athletes).

Przetworzenie i import danych

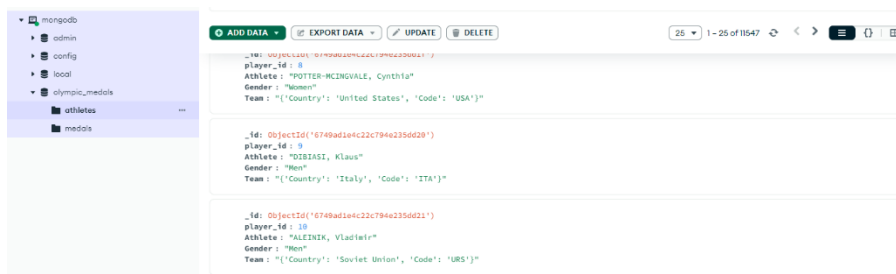
Zdecydowano się na stworzenie trzech kolekcji: games, athletes, medals. Taki podział wydawał się najbardziej logiczny w momencie tworzenia bazy danych ze względu na dostępne dane – takie jak imię, płeć i kraj zawodnika (dla kolekcji athletes), rok i miasto Igrzysk (dla kolekcji games) oraz typ medalu (złoty/srebrny/brąz) i konkurencja w którym został zdobyty (dla kolekcji medals). Ponadto, podział ten umożliwił spełnienie wymagań projektu – stworzenie dokumentów zagnieżdżonych (np. informacje o kraju i kodzie kraju w dokumencie zagnieżdżonym w athletes, informacje o sporcie i wydarzeniu w medals) i referencji (do gry i zawodnika w medals). Zdecydowano się na następujący podział pól:

Athletes	Medals	Games
<ul style="list-style-type: none"> - _id - Athlete (imię) - Gender - Team <ul style="list-style-type: none"> - Team.Country - Team.Code 	<ul style="list-style-type: none"> - _id - Medal - Sport_event <ul style="list-style-type: none"> - Sport_event.Sport - Sport_event.Event - athletes_id - games_id 	<ul style="list-style-type: none"> - _id - Year - City

W dokumentach kolekcji Athletes znajduje się dokument zagnieżdżony Team z polami Country i Code (kraj pochodzenia oraz kod kraju). W dokumentach kolekcji Medals jest dokument zagnieżdżony Sport_event z polami Sport i Event (sport i wydarzenie, np. Pływanie i 300m sprint). W Medals znajdują się też 2 referencje – athletes_id i games_id, wskazujące odpowiednio na zawodnika który wygrał medal i Igrzyska podczas których został zdobyty.

W celu importu danych przez MongoDB Compass, zdecydowano się na stworzenie trzech osobnych plików JSON, po jednym dla każdej kolekcji. Skorzystano w tym celu z języka Python i biblioteki pandas. Podzielono oryginalne dane na trzy kolekcje, dodano dokumenty zagnieżdżone i odpowiednie referencje. Kod użyty w tym celu znajduje się w załączonym pliku prepare_data.py.

Następnie, zaimportowano dane przez MongoDB Compass.



Dokumenty zagnieżdżone i referencje

Dokumenty zagnieżdżone to dokumenty zawarte w dokumentach. Dokument zewnętrzny może posiadać kilka/kolekcję dokumentów zagnieżdżonych – np. w moim przypadku, gdyby zawodnik zmienił obywatelstwo (lub zmieniła się sytuacja geopolityczna, np. upadek ZSSR), miałby 2

dokumenty zagnieżdżone Team, odpowiednio dla starego i nowego obywatelstwa. Referencje umożliwiają na przechowywanie związanych ze sobą informacji w osobnych dokumentach, z polem które reprezentuje tę relację (w moim przypadku np. athletes_id które odnosi się do id właściciela medalu).

Możliwe jest wykonanie zapytania po dokumentach zagnieżdżonych:

```
> db.medals.find(
  { "Sport_event": { "Sport": "Aquatics", "Event": "400m freestyle" } },
  { "Medal": 1, "athletes_id": 1, "_id": 0 }
)
< {
  Medal: 'Silver',
  athletes_id: 14
}
{
  Medal: 'Gold',
  athletes_id: 22
}
{
  Medal: 'Gold',
  athletes_id: 19
}
{
  Medal: 'Bronze',
  athletes_id: 39
}
{
  Medal: 'Silver',
  athletes_id: 13
}
}
```

Podajemy wtedy wartości pól (jednego lub kilku, nie muszą być wszystkie) w zapytaniu. Powyższe zapytanie zwraca medale związane z podanym sportem i wydarzeniem sportowym.

Możliwe są też zapytania po referencjach:

```
> var playerId = db.athletes.findOne({ "Athlete": "PHELPS, Michael" })._id;
db.medals.find(
  { "athletes_id": playerId,
    { "Medal": 1, "Sport_event": 1, "_id": 0 }
  });
< {
  Medal: 'Silver',
  Sport_event: {
    Sport: 'Athletics',
    Event: '200m'
  }
}
{
  Medal: 'Gold',
  Sport_event: {
    Sport: 'Athletics',
    Event: '4x400m relay'
  }
}
{
  Medal: 'Silver',
  Sport_event: {
    Sport: 'Athletics',
    Event: '200m'
  }
}
}
olympic_medals>
```

W powyższym zapytaniu, zapisujemy id danego zawodnika (Micheal Phelps'a) do zmiennej i używamy tej zmiennej by wyszukać po referencji w kolekcji Medals, aby zobaczyć wszystkie medale zawodnika.

Logika biznesowa

Dodano szereg funkcji do dodawania i modyfikacji zmiennych, jak i sekwencję do autoinkrementacji oraz agregacje.

Sekwencja do autoinkrementacji

MongoDB nie posiada wbudowanej sekwencji do autoinkrementacji. Autoinkrementację zrealizowano zatem za pomocą kolekcji counters i funkcji getNextSequence:

```
1  db.counters.insertMany([
2    { _id: "athletes_id", sequence_value: 11547 },
3    { _id: "medals_id", sequence_value: 15316 },
4    { _id: "games_id", sequence_value: 9 }
5  ]);
6
7  function getNextSequence(sequenceName) {
8    const counter = db.counters.findOneAndUpdate(
9      { _id: sequenceName },
10     { $inc: { sequence_value: 1 } },
11     { returnDocument: "after" }
12   );
13   return counter.sequence_value;
14 };
```

Wstawiono ostatni z istniejących numerów id każdego z dokumentów jako startowy.

Funkcje do wstawiania danych:

```
function addAthlete(name, gender, country, code) {
  validateGender(gender);
  const athleteId = getNextSequence("athletes_id");
  const athlete = {
    _id: athleteId,
    Athlete: name,
    Gender: gender,
    Team: { Country: country, Code: code }
  };
  db.athletes.insertOne(athlete);
  return athlete;
};

function addMedal(medalType, sport, event, athletesId, gameId) {
  validateMedalType(medalType);
  validateAthleteId(athletesId);
  validateGameId(gameId);
  const medalId = getNextSequence("medals_id");
  const medal = {
    _id: medalId,
    Medal: medalType,
    Sport_event: { Sport: sport, Event: event },
    athletes_id: athletesId,
    games_id: gameId
  };
  db.medals.insertOne(medal);
  return medal;
};

function addOlympicGame(year, city) {
  validateYear(year);
  const gameId = getNextSequence("games_id");
  const olympicGame = {
    _id: gameId,
    Year: year,
    City: city
  };
  db.games.insertOne(olympicGame);
  return olympicGame;
};
```

Wywoływane funkcje walidacyjne to zaimplementowane przeze mnie funkcje pomocnicze. Sprawdzają one np. czy zgadza się rodzaj medalu:

```
function validateMedalType(medalType){
  const validMedalTypes = ["Gold", "Silver", "Bronze"];
  if (!validMedalTypes.includes(medalType)) {
    throw new Error(`Invalid medal type: ${medalType}. Medal type must be Gold, Silver or Bronze.`);
  }
  return;
}
```

Jak widać na załączonym zrzucie ekranu, funkcje dodające dane korzystają z zaimplementowanej funkcji do autoinkrementacji getNextSequence w celu wygenerowania kolejnego ID.

Wstawienie nowych Igrzysk:

```
> addOlympicGame(2012, 'London');
< { _id: 10, Year: 2012, City: 'London' }
rs0 [primary] olympic_medals>
```

Wstawienie nowego zawodnika:

```
> addAthlete('ŚWIAȚEK, Iga', 'Women', 'Poland', 'POL');
< {
  _id: 11548,
  Athlete: 'ŚWIAȚEK, Iga',
  Gender: 'Women',
  Team: { Country: 'Poland', Code: 'POL' }
}
rs0 [primary] olympic_medals>
```

Wstawienie nowego medalu:

```
> addMedal('Gold', 'Tennis', 'singles', 11548, 10)
< {
  _id: 15317,
  Medal: 'Gold',
  Sport_event: { Sport: 'Tennis', Event: 'singles' },
  athletes_id: 11548,
  games_id: 10
}
```

Jak widać, funkcje obsługujące wstawianie nowych danych i funkcja obsługująca autoinkrementację działają poprawnie.

Funkcje do modyfikacji danych:

Zaimplementowano sumarycznie 14 funkcji do modyfikacji danych. Zaimplementowano funkcje do modyfikacji każdego z pól (oprócz ID) każdego dokumentu i funkcje do modyfikacji wszystkich pól na raz (bez ID).

```
> function updateAthleteCountry(athleteId, newCountry) { ...
}

function updateAthleteCode(athleteId, newCode) {
  const result = db.athletes.updateOne(
    { _id: athleteId },
    { $set: { "Team.Code": newCode } }
  );
  return result.modifiedCount > 0
    ? `Athlete code updated to ${newCode}`
    : `No athlete found with ID ${athleteId}`;
}

function updateAthlete(athleteId, newName, newGender, newCountry, newCode) {
  validateGender(newGender);
  const result = db.athletes.updateOne(
    { _id: athleteId },
    { $set: {
      Athlete: newName,
      Gender: newGender,
      "Team.Country": newCountry,
      "Team.Code": newCode
    }}
  );
  return result.modifiedCount > 0
    ? `Athlete updated`
    : `No athlete found with ID ${athleteId}`;
}

function updateGameYear(gameId, newYear) {
  validateYear(newYear);
  const result = db.games.updateOne(
    { _id: gameId },
    { $set: { Year: newYear } }
  );
  return result.modifiedCount > 0
    ? `Olympic Game year updated to ${newYear}`
    : `No game found with ID ${gameId}`;
}

> function updateGameCity(gameId, newCity) { ...
}

> function updateGame(gameId, newYear, newCity) { ...
}

> function updateMedalType(medalId, newMedalType) { ...
```

W odpowiednich funkcjach modyfikujących dane użyto również wcześniej zaimplementowane funkcje pomocnicze od walidacji danych. Przykłady użycia:

```
> updateAthlete(11548, 'SWIATEK, Aga', 'Women', 'Lithuania', 'LTU')
< Athlete updated
> db.athletes.findOne({_id: 11548})
< {
  _id: 11548,
  Athlete: 'SWIATEK, Aga',
  Gender: 'Women',
  Team: {
    Country: 'Lithuania',
    Code: 'LTU'
  }
}
rs0 [primary] olympic_medals>
```

```
> updateGameCity(10, 'New York')
< Olympic Game city updated to New York
> db.games.findOne({_id: 10})
< {
  _id: 10,
  Year: 2012,
  City: 'New York'
}
rs0 [primary] olympic_medals>
```

```
> db.medals.findOne({_id: 5510})
< {
  _id: 5510,
  Medal: 'Gold',
  Sport_event: {
    Sport: 'Table Tennis',
    Event: 'singles'
  },
  athletes_id: 4411,
  games_id: 4
}
> updateMedalEvent(5510, 'doubles')
< Medal event updated to doubles
> db.medals.findOne({_id: 5510})
< {
  _id: 5510,
  Medal: 'Gold',
  Sport_event: {
    Sport: 'Table Tennis',
    Event: 'doubles'
  },
  athletes_id: 4411,
  games_id: 4
}
rs0 [primary] olympic_medals>
```

Cały kod użyty do implementacji logiki biznesowej znajduje się w załączonym pliku business_logic.js.

Agregacje

Zaimplementowano trzy agregacje:

1. Agregacja zwracająca sumę medali per zawodnik

```
function getMedalsByAthlete() {
  return db.medals.aggregate([
    {
      $lookup: {
        from: "athletes",
        localField: "athletes_id",
        foreignField: "_id",
        as: "athlete_info"
      }
    },
    { $unwind: "$athlete_info" },
    {
      $group: {
        _id: "$athletes_id",
        athlete_name: { $first: "$athlete_info.Athlete" },
        gender: { $first: "$athlete_info.Gender" },
        medals_count: { $sum: 1 }
      }
    },
    { $sort: { medals_count: -1 } }
  ]);
}
```

2. Agregacja zwracająca sumę medali per kraj

```
function getMedalsByCountryTotal() {
  return db.medals.aggregate([
    {
      $lookup: {
        from: "athletes",
        localField: "athletes_id",
        foreignField: "_id",
        as: "athlete_info"
      }
    },
    { $unwind: "$athlete_info" },
    {
      $group: {
        _id: "$athlete_info.Team.Country",
        total_medals: { $sum: 1 }
      }
    },
    { $sort: { total_medals: -1 } }
  ]);
}
```

3. Agregacja zwracająca ilość medali zawodnika i wydarzenia sportowe podczas których zostały zdobyte

```
function getAthleteMedalsAndSports(){
  return db.medals.aggregate([
    {
      $lookup: {
        from: "athletes",
        localField: "athletes_id",
        foreignField: "_id",
        as: "player_info"
      }
    },
    { $unwind: "$player_info" },
    {
      $group: {
        _id: "$athletes_id",
        athlete_name: { $first: "$player_info.Athlete" },
        gender: { $first: "$player_info.Gender" },
        total_medals: { $sum: 1 },
        sports_events: {
          $addToSet: "$Sport_event"
        }
      }
    },
    { $sort: { total_medals: -1 } }
  ]);
}
```


Agregacje wrzucono w funkcję by ułatwić ich ponowne wywoływanie. Przykłady użycia:

```
> getMedalsByAthlete()
< {
  _id: 8563,
  athlete_name: 'HUANG, Chih Hsiung',
  gender: 'Men',
  medals_count: 16
}
{
  _id: 4595,
  athlete_name: 'THOMPSON, Jenny',
  gender: 'Women',
  medals_count: 12
}
{
  _id: 598,
  athlete_name: 'ANDRIANOV, Nikolay',
  gender: 'Men',
  medals_count: 12
}
{
  _id: 6552,
  athlete_name: 'PLUMENAIL, Lionel',
  gender: 'Men',
  medals_count: 12
}
{
  _id: 2292,
  athlete_name: 'TORRES, Dara',
  gender: 'Women',
  medals_count: 12
}
```

```
> getMedalsByCountryTotal()
< {
  _id: 'United States',
  total_medals: 1856
}
{
  _id: 'Soviet Union',
  total_medals: 1098
}
{
  _id: 'Australia',
  total_medals: 719
}
{
  _id: 'Germany',
  total_medals: 690
}
{
  _id: 'East Germany',
  total_medals: 689
}
{
  _id: 'China',
  total_medals: 627
}
```

```
> getAthleteMedalsAndSports()
< {
  _id: 8563,
  athlete_name: 'HUANG, Chih Hsiung',
  gender: 'Men',
  total_medals: 16,
  sports_events: [
    {
      Sport: 'Aquatics',
      Event: '4x200m freestyle relay'
    },
    {
      Sport: 'Aquatics',
      Event: '200m freestyle'
    },
    {
      Sport: 'Aquatics',
      Event: '4x100m freestyle relay'
    },
    {
      Sport: 'Aquatics',
      Event: '200m butterfly'
    },
    {
      Sport: 'Aquatics',
      Event: '4x100m medley relay'
    }
  ]
}
{
  _id: 2292,
  athlete_name: 'TORRES, Dara',
  gender: 'Women',
  total_medals: 12,
  sports_events: [
    {
      Sport: 'Aquatics',
      Event: '4x100m medley relay'
    },
    {
      Sport: 'Aquatics',
      Event: '50m freestyle'
    },
    {
      Sport: 'Aquatics',
      Event: '100m freestyle'
    },
    {
      Sport: 'Aquatics',
      Event: '100m butterfly'
    },
    {
      Sport: 'Aquatics',
      Event: '4x100m freestyle relay'
    }
  ]
}
```

Kod związany z agregacjami znajduje się w załączonym pliku aggregations.js.

Indeksy

Zaimplementowano sumarycznie 6 indeksów. Dla kolekcji Athletes, dodano indeksy na pola Team.Country oraz Gender. Ze względu na tematykę bazy danych, uznano że zapytania związane z krajami i/lub płcią zawodników mogą być częste. Dla kolekcji Games, dodano jedynie indeks na pole Year. Dla kolekcji Medals dodano indeksy na pola athletes_id, games_id oraz indeks kompozytowy na pola Sport_event.Sport i Sport_event.Event.

W celu dodania indeksów, użyto komendy `db.[nazwa_kolekcji].createIndex()`. Kod z implementacją indeksów znajduje się w załączonym pliku `indexes.js`.

Agregacje i Indeksy

W celu sprawdzenia wydajności zapytań, jako pierwszy przykład obrano zaimplementowane agregacje. Czasy agregacji przed dodaniem indeksów:

Agregacja	Czas (ms)	Screenshot
getMedalsByAthlete	1233	<pre>executionStats: { executionSuccess: true, nReturned: 15316, executionTimeMillis: 1233,</pre>
getMedalsByCountryTotal	1138	<pre>executionSuccess: true, nReturned: 15316, executionTimeMillis: 1138, totalKeysExamined: 0,</pre>
getAthleteMedalsAndSports	1127	<pre>executionStats: { executionSuccess: true, nReturned: 15316, executionTimeMillis: 1127, totalKeysExamined: 0,</pre>

Dodano indeks na pole `athletes_id` w kolekcji Medals za pomocą komendy

```
db.medals.createIndex({ "athletes_id": 1 });
```

i przetestowano najpierw agregację `getAthleteMedalsAndSports`:

```
executionStats: {
  executionSuccess: true,
  nReturned: 15316,
  executionTimeMillis: 1048,
  totalKeysExamined: 0,
```

Czas wykonania został skrócony o około 80 milisekund.

Następnie, przetestowano agregację `getMedalsByAthlete`:

Tylko z indeksem na `athletes_id`:

```
executionStats: {  
  executionSuccess: true,  
  nReturned: 15316,  
  executionTimeMillis: 1053,  
  totalKeysExamined: 0,
```

Z dodanym indeksem na pole Gender w Athletes:

```
executionSuccess: true,  
nReturned: 15316,  
executionTimeMillis: 1149,  
totalKeysExamined: 0,
```

Czas wykonania po dodaniu (tylko) indeksu na athletes_id poprawił się o ok. 180 ms. Co ciekawe, po dodaniu indeksu na Gender, czas wykonania się wydłużył. Dzieje się tak zapewne ponieważ Gender jest wykorzystywane jedynie w sekcji group w agregacji, w której indeksy nie są wykorzystywane – dodatkowy indeks jedynie obciążył zapytanie.

Następnie, przetestowano getMedalsByCountryTotal:

```
executionStats: {  
  executionSuccess: true,  
  nReturned: 15316,  
  executionTimeMillis: 918,  
  totalKeysExamined: 0,
```

Czas wykonania poprawił się o około 200 ms.

Względnie mała poprawa czasów wykonania zwróciła moją uwagę. Po użyciu komendy .explain() okazało się, że agregacje wykorzystują operację COLLSCAN (skanowania wszystkich dokumentów / nie używania indeksów).

```
stage: 'COLLSCAN',  
nReturned: 15316,
```

Etap lookup w agregacji nie korzysta z indeksów innych niż te założone na foreignKey (w moim przypadku, z indeksów domyślnie zaimplementowanych na _id). W etapie group, indeksy nie są wykorzystywane. Są one wykorzystywane za to w etapie match jeśli jest on użyty w agregacji.

Zapytania i indeksy

Napisano szereg zapytań w celu przetestowania wpływu indeksów na ich wydajność.

Testy indeksów na kolekcji Athletes:

Indeksy	Zapytanie	Czas (ms)	Screenshot
Brak indeksu	db.athletes.find({Gender:"Women"})	8	<pre>executionStats: { executionSuccess: true, nReturned: 4269, executionTimeMillis: 8,</pre>
Indeks na Gender	db.athletes.find({Gender:"Women"})	6	<pre>executionStats: { executionSuccess: true, nReturned: 4269, executionTimeMillis: 6,</pre>
Brak indeksu	db.athletes.find({"Team.Country": "United States"})	8	<pre>executionStats: { executionSuccess: true, nReturned: 1369, executionTimeMillis: 8, totalKeysExamined: 0, totalDocsExamined: 11547,</pre>
Indeks na Team.Country	db.athletes.find({"Team.Country": "United States"})	2	<pre>executionStats: { executionSuccess: true, nReturned: 1369, executionTimeMillis: 2, totalKeysExamined: 1369,</pre>
Brak indeksu	db.athletes.find({"Team.Country": "United States", Gender: "Women"})	7	<pre>executionStats: { executionSuccess: true, nReturned: 600, executionTimeMillis: 7, totalKeysExamined: 0,</pre>
Indeks na Gender i Team.Country	db.athletes.find({"Team.Country": "United States", Gender: "Women"})	4	<pre>executionStats: { executionSuccess: true, nReturned: 600, executionTimeMillis: 4, totalKeysExamined: 1369,</pre>
Indeks tylko na Team.Country	db.athletes.find({"Team.Country": "United States", Gender: "Women"})	3	<pre>executionStats: { executionSuccess: true, nReturned: 600, executionTimeMillis: 3, totalKeysExamined: 1369,</pre>

Jak widać w przypadku wszystkich testów, dodanie indeksów przyspieszyło czas wykonania zapytania. Zmiany są w niektórych przypadkach niewielkie (pare milisekund), lecz stosunkowo, czas wykonania skraca się nawet o 75% (w przypadku indeksu na Country i zapytaniu o Country). Co ciekawe, przy zapytaniu związanym i z Country i z Gender, wersja jedynie z indeksem na Country jest szybsza niż wersja z indeksem i na Country i na Gender. Dzieje się tak zapewne dlatego, że Gender ma bardzo mało możliwych wartości i z tego powodu posiadanie tego indeksu obciąża bardziej zapytania niż poprawia ich wydajność.

Testy indeksu na Games:

Indeksy	Zapytanie	Czas (ms)	Screenshot
Brak indeksu	db.games.find({year: 2000})	0	<pre>executionStats: { executionSuccess: true, nReturned: 0, executionTimeMillis: 0, totalKeysExamined: 0,</pre>
Indeks na Year	db.games.find({year: 2000})	0	<pre>executionStats: { executionSuccess: true, nReturned: 0, executionTimeMillis: 0, totalKeysExamined: 0, totalDocsExamined: 9,</pre>

Nie widać poprawy po dodaniu indeksu, lecz kolekcja Games jest tak mała, że nawet bez indeksu czas wykonania został zaokrąglony do 0 ms.

Testy indeksów na Medal:

Indeksy	Zapytanie	Czas (ms)	Screenshot
Brak indeksu	db.medals.find({athletes_id: 10, games_id: 2})	7	<pre>executionStats: { executionSuccess: true, nReturned: 1, executionTimeMillis: 7, totalKeysExamined: 0,</pre>
Indeks na athlete_id i game_id	db.medals.find({athletes_id: 10, games_id: 2})	2	<pre>executionStats: { executionSuccess: true, nReturned: 1, executionTimeMillis: 2, totalKeysExamined: 2,</pre>

Brak indeksu	db.medals.find({athletes_id: 10})	7	executionStats: { executionSuccess: true, nReturned: 2, executionTimeMillis: 7, totalKeysExamined: 0,
Indeks na athlete_id	db.medals.find({athletes_id: 10})	0	executionStats: { executionSuccess: true, nReturned: 1, executionTimeMillis: 0, totalKeysExamined: 1,
Brak indeksu	db.medals.find({games_id: 2})	9	executionStats: { executionSuccess: true, nReturned: 1387, executionTimeMillis: 9, totalKeysExamined: 0,
Indeks na game_id	db.medals.find({games_id: 2})	2	executionStats: { executionSuccess: true, nReturned: 1387, executionTimeMillis: 2, totalKeysExamined: 1387, totalDocsExamined: 1387,
Brak indeksu	db.medals.find({"Sport_event.Sport": "Athletics"})	8	executionStats: { executionSuccess: true, nReturned: 1523, executionTimeMillis: 8, totalKeysExamined: 0,
Tylko indeks na Sport_event.sport	db.medals.find({"Sport_event.Sport": "Athletics"})	12	executionStats: { executionSuccess: true, nReturned: 1523, executionTimeMillis: 12, totalKeysExamined: 1523,
Indeks na Sport_event.sport i Sport_event.event	db.medals.find({"Sport_event.Sport": "Athletics"})	6	executionStats: { executionSuccess: true, nReturned: 1523, executionTimeMillis: 6, totalKeysExamined: 1523,

Jak widać, zastosowanie indeksów skróciło czas wykonywania zapytań. Szczególnie podniósł wydajność indeks na athlete_id przy zapytaniu o athlete_id. Co ciekawe, przy zapytaniu które dotyczyło tylko Sport_event.Sport, indeks kompozytowy na Sport_event.Sport i Sport_event.Event znacznie poprawił wydajność zapytania, podczas kiedy indeks tylko na Sport_event.Sport pogorszył wydajność.

Transakcja

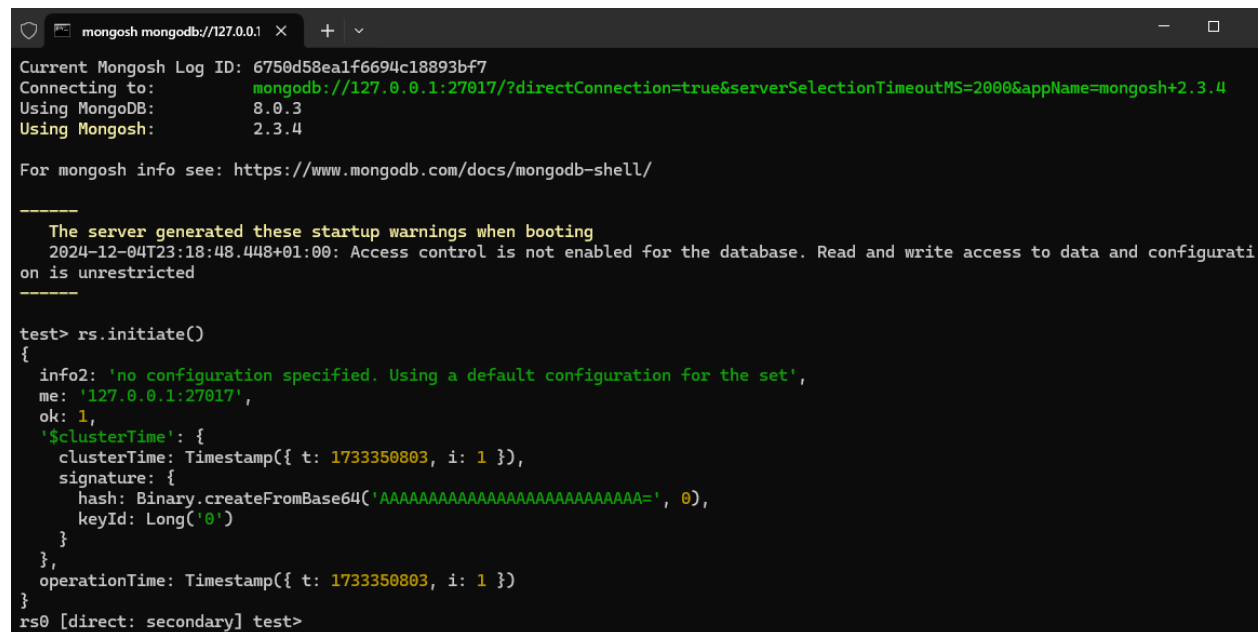
Przygotowanie bazy danych

Użycie transakcji wymaga, by docelowa baza danych była tzw replica set. Replica set to grupa procesów mongodb które posiadają te same dane (wykorzystywana jest głównie w celu zapewnienia redundancji i dostępności). Dla przykładu wystarczy, że moja baza danych będzie zbiorem replik z jedną repliką. Ponieważ stworzyłam bazę jako tzw standalone self-managed, musiałam wprowadzić pewne modyfikacje:

W mongod.cfg, dodałam:

```
replication:
  replSetName: rs0
```

A następnie zrestartowałam MongoDB komendami net stop MongoDB i net start MongoDB. Następnie w konsoli mongosh, wykonałam komendę rs.initiate().



```
mongosh mongodb://127.0.0.1
Current Mongosh Log ID: 6750d58e1f6694c18893bf7
Connecting to:
  mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.4
Using MongoDB:
  8.0.3
Using Mongosh:
  2.3.4

For mongosh info see: https://www.mongodb.com/docs/mongosh-shell/

-----
The server generated these startup warnings when booting
2024-12-04T23:18:48.448+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----

test> rs.initiate()
{
  info2: 'no configuration specified. Using a default configuration for the set',
  me: '127.0.0.1:27017',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1733350803, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAAA', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1733350803, i: 1 })
}
rs0 [direct: secondary] test>
```

Po tym kroku, moja baza danych była przygotowana na obsługę transakcji.

Przykład

Transakcje zapewniają, że operacje na dokumentach będą atomowe – tzn, że wykonana będą w całości lub w ogóle. Jeśli dojdzie do „commit’u” transakcji, wszystkie zmiany na danych zostaną zapisane. Jeśli dojdzie do „abort’u” transakcji, wszystkie zmiany zostaną odrzucone, i baza pozostanie w stanie sprzed transakcji.

MongoDB umożliwia wykonanie transakcji za pomocą np. API transakcji. API jest dostępne w kilku językach programowania – w moim przykładzie, używam Node.js.

Wykonanie transakcji składa się z kilku etapów. Należy najpierw stworzyć instancję klienta połączoną z docelową bazą danych.

```
const { MongoClient } = require('mongodb');

async function main(){
  const client = new MongoClient("mongodb://localhost:27017");

  try {
    await client.connect();

    await performTransaction(client);
  } finally {
    await client.close();
  }
}
```

W moim przypadku, łączyłam się z bazą na localhost:27017. Następnie, należy wybrać kolekcję (lub zbiór kolekcji) na których przeprowadzone będą operacje i należy stworzyć sesję.

```
async function performTransaction(client){
  const medals = client.db('olympic_medals').collection('medals');
  const session = client.startSession();
```

Można, ale nie trzeba, zdefiniować opcje transakcji:

```
const transactionOptions = {
  readPreference: 'primary',
  readConcern: { level: 'local' },
  writeConcern: { w: 'majority' }
};
```

readPreference oznacza do którego węzła w replica set powinny być skierowane odczyty w transakcji. readConcern określa poziom izolacji dla operacji read. writeConcern dotyczy warunków, które określają czy operacja przebiegła pomyślnie.

Następnie, korzystając z sesji i stworzonej transakcji, należy wykonać nasze operacje na danych:

```
try {
  await session.withTransaction(async () => {
    const res = await medals.updateOne(
      { _id: 3 },
      {
        $set: {
          "Sport_event.Sport": "Swimming",
          "Sport_event.Event": "200m Butterfly"
        }
      },
      { session }
    );
    console.log(`Result of transaction: ${res.modifiedCount}`);
    const updatedMedal = await medals.findOne({ "Sport_event.Sport": "Swimming", {session} });
    console.log("Edited medal ID: " + updatedMedal._id);
    if (!updatedMedal) {
      throw new Error("Medal not found.");
    }

    session.commitTransaction();
    console.log("Transaction successful");
  }, transactionOptions);
}
```


Jak widać na zamieszczonym zrzucie ekranu, wszystkie operacje na danych są wykonane w bloku `withTransaction` – czyli wykorzystujemy transakcję do ich przeprowadzenia. Następnie, należy zrobić `commit` transakcji. Jeśli wystąpi błąd, dokonujemy `abort` transakcji – wszystkie zmiany są odrzucane i wracamy do stanu sprzed transakcji:

```
} catch (error) {  
  console.log("Transaction error:", error);  
  await session.abortTransaction();  
}  
finally {  
  await session.endSession();  
}
```

Pod koniec, należy skończyć sesję i zamknąć połączenie klienta z bazą.

Przykład użycia – dane sprzed wykonania transakcji:



Wykonanie transakcji:

```
C:\Users\magma\Desktop\zaawansowane bazy danych\etap III>node transaction.js  
Result of transaction: 1  
Edited medal ID: 3  
Transaction successful  
  
C:\Users\magma\Desktop\zaawansowane bazy danych\etap III>
```

Dane po wykonaniu transakcji:



Podsumowując, podczas transakcji w MongoDB, należy otworzyć sesję kliencką, w bloku transakcji przeprowadzić operacje na danych, a następnie przeprowadzić `commit` lub `abort` transakcji, w zależności czy wystąpił błąd. Pod koniec, należy zamknąć sesję oraz połączenie.

Cały kod został umieszczony w załączonym pliku `transaction.js`.

Źródła / wykorzystane materiały pomocnicze:

<https://www.mongodb.com/docs/manual/core/transactions/>

<https://www.mongodb.com/docs/manual/tutorial/convert-standalone-to-replica-set/>

przykłady z <https://github.com/mongodb-developer/nodejs-quickstart>

Zaawansowane funkcjonalności – zapytania geoprzestrzenne

Ze względu na pole City w kolekcji dokumentów Games (oznaczające miasto w którym odbywały się Igrzyska Olimpijskie), zdecydowano się na implementację zapytań geoprzestrzennych.

Zapytanie geoprzestrzennie (Geospatial Queries) to funkcjonalność pozwalająca na przeprowadzenie operacji na danych geoprzestrzennych (np. na długości i szerokości geograficznej). Zapytania pozwalają na znalezienie dokumentów na podstawie ich zależności geograficznej (/zależności ich danych geoprzestrzennych) w stosunku do np. określonych miejsc lub obszarów – może np. wykonać zapytanie związane z odległością od danego punktu lub znajduwaniem się (lub nie) w danym obszarze.

Dane geoprzestrzenne są przechowywane w formie GeoJSON, jako np. Point lub Polygon. Dlatego też w ramach implementacji, dodano pole location do dokumentów, o typie Point i szerokości i długości geograficznej danego miasta.

```
> db.games.updateOne(
  { City: "Montreal" },
  { $set: { location: { type: "Point", coordinates: [-73.561668, 45.508888] } } } // long, lat
);

db.games.updateOne(
  { City: "Moscow" },
  { $set: { location: { type: "Point", coordinates: [37.618423, 55.751244] } } }
);

db.games.updateOne(
  { City: "Los Angeles" },
  { $set: { location: { type: "Point", coordinates: [-118.243683, 34.052235] } } }
);

db.games.updateOne(
  { City: "Seoul" },
  { $set: { location: { type: "Point", coordinates: [127.024612, 37.532600] } } }
);
```

Plik z całym kodem związanym z dodaniem pola location znajduje się w załączonym pliku geoqueries.js.

Funkcjonalność Geospatial Queries umożliwia też dodanie indeksu typu „2dsphere” na pole związane z danymi geoprzestrzennymi. Indeks 2dsphere wspiera zapytania związane z geometrią kuli ziemskiej.

```
> db.games.createIndex({ location: "2dsphere" });  
< location_2dsphere  
rs0 [primary] olympic_medals>
```

Wykonano dwa zapytania w celu przetestowania tej funkcjonalności.

Zapytanie 1

Pierwsze zapytanie dotyczyło Igrzysk, które odbyły się maksymalnie 1200km od Warszawy:

```
1 const { MongoClient } = require("mongodb");  
2  
3 async function findNearbyGames() {  
4   const client = new MongoClient("mongodb://localhost:27017");  
5   try {  
6     await client.connect();  
7  
8     const db = client.db("olympic_medals");  
9     const games = db.collection("games");  
10  
11     const nearbyGames = await games.find({  
12       location: {  
13         $near: {  
14           $geometry: { type: "Point", coordinates: [21.017532, 52.237049] },  
15           $maxDistance: 1200000  
16         }  
17       }  
18     }).toArray();  
19  
20     console.log("Nearby Games:", nearbyGames);  
21   } finally {  
22     await client.close();  
23   }  
24 }  
25  
26 findNearbyGames().catch(console.error);  
27
```

```
C:\Users\magma\Desktop\zaawansowane bazy danych\etap III>node geoqueries_query1.js  
Nearby Games: [  
  {  
    _id: 2,  
    Year: 1980,  
    City: 'Moscow',  
    location: { type: 'Point', coordinates: [Array] }  
  }  
]  
  
C:\Users\magma\Desktop\zaawansowane bazy danych\etap III>
```

Takie Igrzyska odbyły się w Moskwie w 1980. Pole \$near pozwala na znalezienie dokumentu z danymi geoprzestrzennymi w określonej odległości od danego punktu. W powyższym przykładzie, punktem tym była Warszawa i podano jedynie maksymalną odległość (można też w takim zapytaniu podać np. minimalną odległość). Kod znajduje się w załączonym pliku geoqueries_query1.js.

Zapytanie 2

Drugie zapytanie dotyczyło Igrzysk Olimpijskich które odbyły się w podanym obszarze.

```
1 const { MongoClient } = require("mongodb");  
2  
3 async function findGamesInBoundary() {  
4   const client = new MongoClient("mongodb://localhost:27017");  
5   try {  
6     await client.connect();  
7  
8     const db = client.db("olympic_medals");  
9     const games = db.collection("games");  
10  
11     const gamesInBoundary = await games.find({  
12       location: {  
13         $geoWithin: {  
14           $box: [  
15             [-10.0, 35.0], // South-West  
16             [30.0, 70.0] // North-East  
17           ]  
18         }  
19       }  
20     }).toArray();  
21  
22     console.log("Games within boundaries:", gamesInBoundary);  
23   } finally {  
24     await client.close();  
25   }  
26 }  
27  
28 findGamesInBoundary().catch(console.error);  
29
```

```
C:\Users\magma\Desktop\zaawansowane bazy danych\etap III>node geoqueries_query2.js  
Games within boundaries: [  
  {  
    _id: 5,  
    Year: 1992,  
    City: 'Barcelona',  
    location: { type: 'Point', coordinates: [Array] }  
  },  
  {  
    _id: 8,  
    Year: 2004,  
    City: 'Athens',  
    location: { type: 'Point', coordinates: [Array] }  
  }  
]  
  
C:\Users\magma\Desktop\zaawansowane bazy danych\etap III>
```

W obszarze określonym za pomocą pola \$box odbyły się dwa Igrzyska – w Barcelonie w 1992 i w Atenach w 2004. Skorzystano z opcji zapytania \$geoWithin, która pozwala na znalezienie

dokumentów z danymi przestrzennymi które znajdują się w określonym obszarze. Kod znajduje się w załączonym pliku geoqueries_query2.js.

Funkcjonalność zapytań geoprzestrzennych wydaje się być bardzo przydatna jeśli baza danych posiada dane geoprzestrzenne i przewidujemy zapytania związane np. z odległością. Zapytania są bardzo łatwe do zaimplementowania – wystarczy użyć np. \$near a baza danych sama zajmie się resztą.

Źródła:

Szerokości / długości geograficzne brano z <https://www.latlong.net/>

Informacje pochodzą z <https://www.mongodb.com/docs/manual/geospatial-queries/>

Wnioski

Baza dokumentowa nadawała się do wybranego zbioru danych, lecz baza relacyjna byłaby potencjalnie lepsza ze względu na bezpieczeństwo integralności danych.

Baza dokumentowa zapewnia więcej elastyczności – pozwala np. na posiadanie dokumentów zagnieżdżonych. Dokumenty zagnieżdżone pozwalają na logiczne zorganizowanie danych i łatwy dostęp do nich. W bazie relacyjnej, zapewne byłyby one osobną tabelą. Ponadto, bazy dokumentowe są lepiej przystosowane do zmiany w strukturze dokumentów i pozwalają np. żeby tylko niektóre dokumenty w kolekcji miały dodatkowe pola.

Z drugiej strony, baza relacyjna zapewnia dużo lepsze bezpieczeństwo integralności danych. W MongoDB nie istnieją foreign key constraints które by można użyć w referencjach, co mogłoby spowodować większą ilość błędów w bazie. W wypadku implementacji wybranego zbioru w bazie relacyjnej, należałoby stworzyć odpowiednie constrainty które dbałyby o integralność relacji między np. athlete a medal. Ponadto, bazy relacyjne lepiej obsługują związki między danymi niż np. referencje (między np. athletes a medals) – operacje lookup w MongoDB są mniej wydajne niż JOINy. SQL też lepiej sobie radzi z złożonymi zapytaniami wymagającymi wiele operacji JOIN.

Bazy dokumentowe są często ‘bardziej nastawione’ na denormalizację danych. Jest to potencjalnie wygodniejsze dla programisty ponieważ brak potrzeby normalizacji upraszcza proces projektowania bazy, i denormalizacja może też pomóc w szybszym wyszukiwaniu danych. Z drugiej strony, może prowadzić do zbytnej redundancji danych i większej ilości błędów. W przypadku implementacji bazy dla wybranego problemu jako bazy relacyjnej, należałoby przeprowadzić normalizację. W ramach normalizacji, należałoby np. podzielić imię zawodników na imię i nazwisko, zadbać o niezależność pól niekluczowych od innych niekluczowych w przypadku np. kraju i kodu kraju, lub np. stworzyć enum dla typów medali.