Nowoczesne metody przetwarzania danych

Serwis społecznościowy à la Tinder

Projekt zaliczeniowy — sprawozdanie

Autor: **Jakub Kurczewski**
Nr Indeksu: **24555**

Spis treści

Link do repozytorium projektu (w repozytorium kod, który pozwoli na stworzenie bazy danych oraz aplikacja Node.js+Express.js odpowiedzialna za wystawienie API)
https://github.com/jkurczewski/s6-nmpd

1. Skrócony opis projektu

Pomysł na projekt powstał na bazie popularnej aplikacji do randkowania Tinder i jest swoistym rozwinięciem zaproponowanej tam koncepcji relacji społecznych. W stworzonym modelu nacisk został postawiony na utrzymanie użytkownika i odejście od jedynie romantycznej zasady działania aplikacji. Koncept projektu przewiduje możliwość łączenia ludzi na kilku poziomach: znajomi, para, znajoma para. Takie rozwiązanie otwiera przed użytkownikiem szanse na poznanie nie tylko miłości swojego życia, a także nowych znajomych, a nawet poznania się z innymi parami.

2. Opis wybranego modelu danych wraz z uzasadnieniem

a. Model danych

Naturalnym modelem danych myśląc o relacjach społecznych pomiędzy osobami, jest rozwiązanie oparte na modelu grafowym. Taki model bazuje na relacjach wzajemnie pomiędzy użytkownikami, a także innymi strukturami społecznymi.

b. Rozwiązanie bazodanowe

Bazując na potrzebach projektu, wybór padł na bazę danych **NoSQL - Neo4j.**

c. Uzasadnienie wyboru

Uzasadnienie wyboru można podzielić na dwie części: wybór pomiędzy bazami SQL a NoSQL, a następnie wybór konkretnej technologii.

Jest wiele czynników, które stanowią o wyższości rozwiązań NoSQL w kontekście klasycznego podejścia do przechowywania danych opartych o SQL. Głównymi punktami, na podstawie których dokonano wyboru były:

a) o wiele lepsza możliwość skalowania baz typu NoSQL w porównaniu do baz typu SQL — w bazach typu NoSQL rezygnuje się ze spójności na rzecz większej wydajności i tolerancji na partycje (**Zgodnie z teorią CAP)**
b) lepsze przystosowanie baz danych NoSQL do obsługi i analizy BigData
c) brak narzuconych modeli danych — danych nie musimy przechowywać w tabelach, co pozwala lepiej dopasować rozwiązanie do problemu, na który ma odpowiadać baza danych

W ramach uzasadnienia wyboru technologii wyszczególniono następujące punkty, które są charakterystyczne dla omówionego modelu danych oraz bazy danych:

a) naturalne dla relacji społecznych jest rozwiązanie oparte na węzłach i relacjach między nimi;
b) dzięki dobremu dopasowaniu przedstawianych danych do modelu skrócił się znacząco czas wdrożenia — baza jest naturalnie przygotowana do obsługi takiego rozwiązania
c) struktura danych oparta o grafy otwiera drzwi przed dużą bazą algorytmów służących do analizowania zgromadzonych treści — łatwość i dostępność funkcji do Data Science jest bardzo rozbudowaną częścią Neo4j

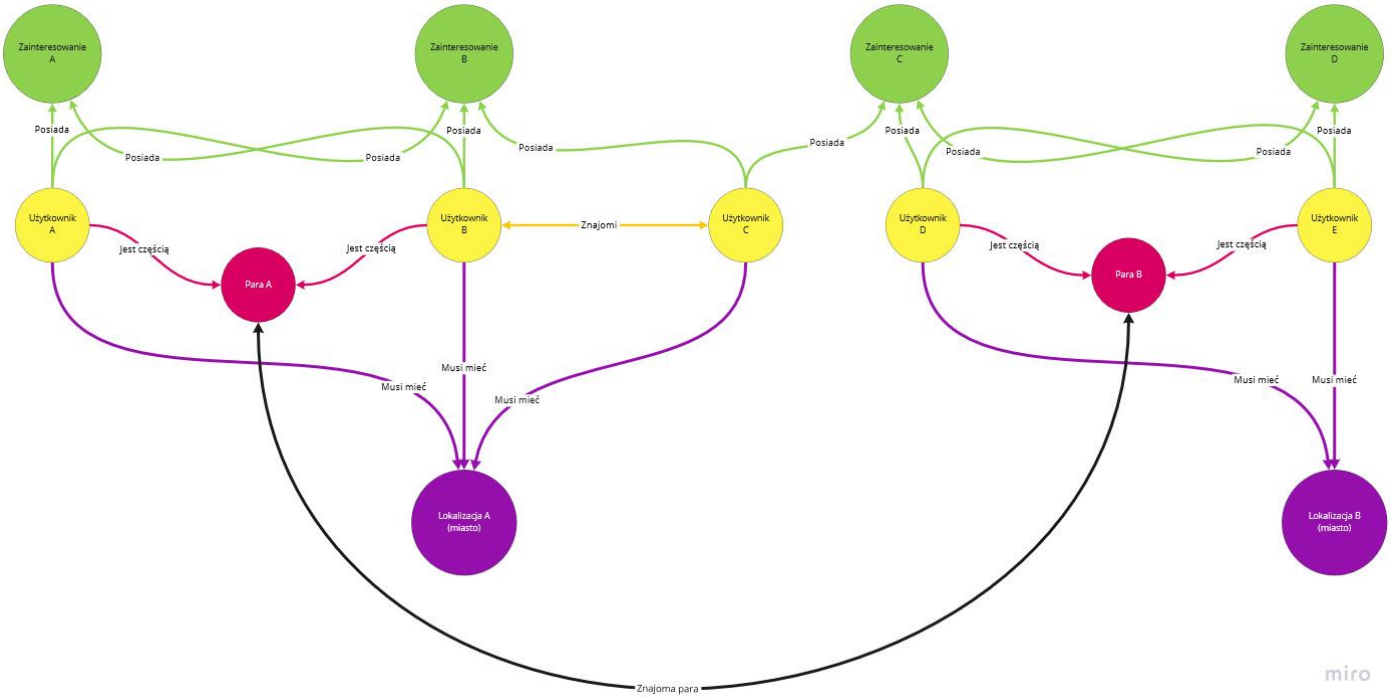
3. Szczególne własności bazy danych wykorzystane w projekcie

Neo4j udostępnia wiele algorytmów, które są unikatowe dla rozwiązań grafowych. Wiele z nich ma zastosowanie w sieciach społecznościowych, zwłaszcza w „data science”. Choć zakres danych prezentowany w projekcie jest niewielki, pozwoliło to na użycie dwóch algorytmów, które czynią projekt o wiele ciekawszym:

1. Algorytmy typu Centrality pozwolą na określenie najbardziej popularnych zainteresowań wśród użytkowników;
2. Algorytmy typu Similarity pozwolą na skuteczne proponowanie użytkownikom nowych znajomcyh, potencjalnych partnerów, a nawet do wyszukiwania podobnych par do posiadanej.

4. Opis implementacji

a. Schemat bazy danych



Legenda:

zielone koła — zainteresowania użytkowników
zielone linie — relacje użytkowników z zainteresowaniami (1 użytkownik może posiadać wiele zainteresowań; użytkownik nie może być w dwóch relacjach z jednym zainteresowaniem)
żółte koła — użytkownicy
żółte linie — relacja znajomości pomiędzy użytkownikami (użytkownicy mogą być znajomymi)
fioletowe koła — lokalizacje (miasta)
fioletowe linie — relacje użytkowników z lokalizacjami (każdy użytkownik musi mieć przypisaną lokalizację)
różowe koła — pary (para musi składać się z dwóch użytkowników i nie może bez nich istnieć)
różowe linie — relacja użytkownika do pary (użytkownik może mieć tylko jedną parę w danym momencie)
czarna linia — relacje znajomości pomiędzy dwiema parami

b. Opis zaimplementowanych metod

i. Pokaż wszystkich użytkowników

Metoda listuje wszystkich użytkowników znajdujących się w bazie danych.

```
const findAll = async () => {
  const session = driver.session({ database });

  try {
    const res = await session.run(
      `MATCH
          (u :User)
          WITH
          {
              id: id(u),
              name: u.name,
              age: u.age,
              sex: u.sex
            } as User
            RETURN
            User`
    );

    return res.records.map((u) => u.get("User"));
  } catch (e) {
    return e;
```

```
  } finally {
    session.close();
  }
};
```

ii. Pokaż wybranego użytkownika

Metoda pokazuje wybranego użytkownika. Filtrowanie dokonane po ID.

```
api.get("/users/:id", async (req, response) => {
  const session = driver.session({ database });

  session
    .run(
      `MATCH
        (u:User) -[r]-> (n)
        WHERE
        id(u) = ${req.params.id}
        RETURN
        u, r, n`
    )
    .then((result) => {
      const user = result.records[0].get("u").properties;
      const nodes = result.records.map((u) => u.get("n"));
      const relation = result.records.map((u) => u.get("r"));

      response.json({ user, relation, nodes });
    })
    .catch((error) => {
      response.status(500).send({
        success: false,
        message: error.message,
      });
    })
```

```
    .finally(() => {
      session.close();
    });
});
```

iii. Wyszukaj podobne pary do podanej pary

Metoda działa w ramach poniższego algorytmu:

1. Wyszukanie i zebranie danych nt. użytkownika wzorcowego (SEEKING_USER)
2. Wyszukanie, zebranie i przefiltrowanie danych nt. pozostałych użytkowników (WANTED_USERS)
3. Rzutowanie danych oraz uruchomienie funkcji służącej do analizy podobieństwa
4. Zwrot rezultatów od najbardziej podobnej osoby do najmniej

```
api.get("/partners/user/:id", async (req, response) => {
  const session = driver.session({ database });

  session
    .run(
      ` MATCH
            (HOBBY :Hobby) <-[:LIKE]- (SEEKING_USER :User)-[:LIVE_IN]->(CITY)
        WHERE
            id(SEEKING_USER) = ${req.params.id}
        WITH
            SEEKING_USER,
            CITY,
            collect(id(HOBBY)) AS SEEKING_USER_HOBBY_IDS

        MATCH
            (WANTED_USERS_HOBBY :Hobby) <-[:LIKE]- (WANTED_USERS :User)
-[:LIVE_IN]->(CITY)
        WHERE
            WANTED_USERS.age < SEEKING_USER.age+5 OR WANTED_USERS.age >
SEEKING_USER.age-5
        WITH
```

```
            SEEKING_USER,
            SEEKING_USER_HOBBY_IDS,
            WANTED_USERS,
            collect(id(WANTED_USERS_HOBBY)) AS WANTED_USERS_HOBBY_IDS
        WITH
            WANTED_USERS,
            id(WANTED_USERS) as WANTED_USERS_IDS,
            SEEKING_USER,
            [id in SEEKING_USER_HOBBY_IDS WHERE id in WANTED_USERS_HOBBY_IDS]
as COMMON_HOBBIES_IDS,
            gds.similarity.overlap(SEEKING_USER_HOBBY_IDS,
WANTED_USERS_HOBBY_IDS)*100 AS SIMILARITY
        ORDER BY
            SIMILARITY DESC
        WHERE
            SIMILARITY > 50 AND SEEKING_USER.sex <> WANTED_USERS.sex
        WITH
            {
                POSSIBLE_FRIEND_ID: WANTED_USERS_IDS,
                COMMON_HOBBIES_IDS: COMMON_HOBBIES_IDS,
                SIMILARITY: SIMILARITY
            } AS Result
        RETURN
            Result
    `
    )
    .then((result) => {
      response.json(result.records.map((r) => r.get("Result")));
    })
    .catch((error) => {
      response.status(500).send({
        success: false,
        message: error.message,
      });
    })
    .finally(() => {
      session.close();
    });
});
```

iv. Wyszukaj podobne pary do podanej pary

Metoda działa w ramach poniższego algorytmu:

1. Wyszukanie i zebranie danych nt. pary wzorcowej. Do zebrania danych wykorzystane jest ID jedynie jednego z użytkowników, który znajduje się w relacji typu PARA z innym użytkownikiem.
2. Wyszukanie, zebranie i przefiltrowanie danych nt. pozostałych par
3. Rzutowanie danych oraz uruchomienie funkcji służącej do analizy podobieństwa
4. Zwrot rezultatów z sortowaniem od największego podobieństwa

```
api.get("/partners/pair/:id", async (req, response) => {
  const session = driver.session({ database });

  session
    .run(
      ` MATCH
            (HOBBY_1 :Hobby) <-[:LIKE]- (USER_1 :User) -[:IN_RELATIONSHIP]->
(RELATION) <-[:IN_RELATIONSHIP]- (:User) -[:LIKE]-> (HOBBY_2 :Hobby)
        WHERE
            id(USER_1) = ${req.params.id}
        WITH
            collect(HOBBY_1)+collect(HOBBY_2) as HOBBIES_RAW,
            RELATION
        UNWIND
            HOBBIES_RAW as COMMON_HOBBIES_RAW
        WITH
            distinct(id(COMMON_HOBBIES_RAW)) as COMMON_HOBBIES_IDS, RELATION
        MATCH
            (HOBBY_3 :Hobby) <-[:LIKE]- (:User) -[:IN_RELATIONSHIP]->
(RELATION_2) <-[:IN_RELATIONSHIP]- (:User) -[:LIKE]-> (HOBBY_4 :Hobby)
        WHERE
            RELATION <> RELATION_2
```

```
        WITH
            collect(HOBBY_3)+collect(HOBBY_4) as HOBBIES_RAW,
            RELATION_2,
            RELATION,
            COMMON_HOBBIES_IDS
        UNWIND
            HOBBIES_RAW as COMMON_HOBBIES_RAW
        WITH
            RELATION,
            RELATION_2,
            collect(distinct COMMON_HOBBIES_IDS) as SEEKING_COUPLE_HOBBY_IDS,
            collect(distinct id(COMMON_HOBBIES_RAW)) as
WANTED_COUPLE_HOBBY_IDS
        WITH
            id(RELATION_2) as WANTED_COUPLE_ID,
            [id in WANTED_COUPLE_HOBBY_IDS WHERE id in
SEEKING_COUPLE_HOBBY_IDS] as COMMON_HOBBIES_IDS,
            gds.similarity.overlap(SEEKING_COUPLE_HOBBY_IDS,
WANTED_COUPLE_HOBBY_IDS)*100 AS SIMILARITY_LEVEL
        ORDER BY
            SIMILARITY_LEVEL DESC
        WITH {
            WANTED_COUPLE_ID: WANTED_COUPLE_ID,
            COMMON_HOBBIES_IDS: COMMON_HOBBIES_IDS,
            SIMILARITY_LEVEL: SIMILARITY_LEVEL
            } as Result
        RETURN
            Result

      `
    )
    .then((result) => {
      response.json(result.records.map((r) => r.get("Result")));
    })
    .catch((error) => {
      response.status(500).send({
        success: false,
        message: error.message,
      });
    })
    .finally(() => {
      session.close();
    });
```

```
});
```

v. Pokaż listę wszystkich hobby posortowaną w zależności od popularności

Metoda pokazuje posortowaną listę wszystkich hobby w systemie w zależności od popularności wśród użytkowników

```
api.get("/hobbies", async (req, response) => {
  const session = driver.session({ database });

  session
    .run(
      ` CALL
          gds.degree.stream('hobbies')
        YIELD
          nodeId,
          score
        WITH
          id(gds.util.asNode(nodeId)) AS HOBBY_ID,
          gds.util.asNode(nodeId).name AS HOBBY,
          score AS FOLLOWERS
        ORDER BY
          FOLLOWERS DESC
        WITH {
          HOBBY_ID: HOBBY_ID,
          HOBBY: HOBBY,
          FOLLOWERS: FOLLOWERS } as Result
        RETURN
          Result
        LIMIT
          10;
      `
```

```
    )
    .then((result) => {
      response.json(result.records.map((r) => r.get("Result")));
    })
    .catch((error) => {
      response.status(500).send({
        success: false,
        message: error.message,
      });
    })
    .finally(() => {
      session.close();
    });
});
```

vi. Pokaż wszystkich użytkowników posiadających dane hobby

Metoda zwraca listę użytkowników, którzy są w relacji z podanym hobby

```
api.get("/hobbies/:name", async (req, response) => {
  const session = driver.session({ database });

  session
    .run(
      ` MATCH
            (USER :User) -[:LIKE]-> (HOBBY :Hobby)
        WHERE
            HOBBY.name = '${req.params.name}'
        WITH {
            id: id(USER),
            name: USER.name,
            age: USER.age,
            sex: USER.sex } as Result
        RETURN
            Result
      `
    )
    .then((result) => {
      response.json(result.records.map((r) => r.get("Result")));
    })
    .catch((error) => {
      response.status(500).send({
        success: false,
```

```
        message: error.message,
      });
    })
    .finally(() => {
      session.close();
    });
});
```

vii. Stwórz relację znajomości pomiędzy dwoma użytkownikami

Metoda tworzy relację znajomości pomiędzy dwoma podanymi użytkownikami.

```
api.post("/friends/:us1&:us2", async (req, response) => {
  const session = driver.session({ database });

  session
    .run(
      ` MATCH
            (USER_1 :User),
            (USER_2 :User)
        WHERE
            id(USER_1) = ${req.params.us1} AND id(USER_2) = ${req.params.us2}
        MERGE
            (USER_1) -[r :FRIEND_TO]-> (USER_2)
        WITH
        {
            USER_1: {
                id: id(USER_1), name: USER_1.name, age: USER_1.age,
                sex: USER_1.sex },
            RELATION: type(r),
            USER_2: {
                id: id(USER_2), name: USER_2.name, age: USER_2.age,
                sex: USER_2.sex }
        } as Result
        RETURN
            Result
      `
    )
    .then((result) => {
```

```
      response.json(result.records.map((r) => r.get("Result")));
    })
    .catch((error) => {
      response.status(500).send({
        success: false,
        message: error.message,
      });
    })
    .finally(() => {
      session.close();
    });
})
```

viii. Stwórz parę pomiędzy dwoma użytkownikami

Metoda tworzy parę pomiędzy dwoma podanymi użytkownikami. Metoda, zanim dokona stworzenia relacji sprawdza, czy podani użytkownicy nie są już w innej parze z użytkownikiem.

```
api.post("/relation/:us1&:us2", async (req, response) => {
  const session = driver.session({ database });

  session
    .run(
      ` OPTIONAL MATCH
            (USER_1 :User)-[:IN_RELATIONSHIP]->(RELATION :Relationship)
        WHERE
            id(USER_1) = ${req.params.us1} OR id(USER_1) = ${req.params.us2}
        CALL
            apoc.do.when
                (
                    RELATION IS null ,
                    '
                        MATCH
                            (USER_1 :User),
                            (USER_2 :User)
                        WHERE
                            ID(USER_1) = ${req.params.us1} AND ID(USER_2) =
${req.params.us2}
                        CREATE
                            (USER_1)-[:IN_RELATIONSHIP]->(r
:Relationship)<-[:IN_RELATIONSHIP]-(USER_2)
                        RETURN r AS result
                    ',
```

```
                    'RETURN "One of given person is already in relationship!"
AS result',
                    {}
                ) YIELD value
        RETURN
            value
        `
    )
    .then((result) => {
      response.json(result.records.map((r) => r.get("value")));
    })
    .catch((error) => {
      response.status(500).send({
        success: false,
        message: error.message,
      });
    })
    .finally(() => {
      session.close();
    });
});
```

5. Uwagi końcowe

a. Podsumowanie projektu

Projekt okazał się być wyjątkowo ciekawym wyzwaniem, zwłaszcza w zakresach tworzenia i obsługi bazy danych. Struktura danych, która w ramach rozwiązań SQL przysporzyłaby wiele pracy, okazała się być wyjątkowo prostą, logiczną i składającą się w pełną całość kiedy została wdrożona przy pomocy rozwiązania grafowego. Ponadto dużym zaskoczeniem podczas wdrożenia okazała się prostota analizy danych. Niewielkim nakładem pracy można osiągnąć w takiej strukturze niejednokrotnie skomplikowane wyniki.

Idealnym podsumowaniem pracy nad projektem okazała się być nietrafiona estymacja czasowa, którą przewidziałem na wdrożenie. Biorąc pod uwagę dotychczasowe projekty z podobnym poziomem skomplikowania, założyłem, że do ukończenia projektu będę potrzebować przynajmniej kilkadziesiąt godzin pracy. Dlatego wielkim zdziwieniem była dla mnie dynamika prac, na którą pozwala opisywany model. Projekt udało mi się ukończyć z prawie 60% zapasem czasu. Pozornie skomplikowany proces generowania bazy danych oraz kreowania metod okazał się być zaskakująco prosty. Nawet pomimo braku doświadczenia z technologią oraz przeciętnymi umiejętnościami programistycznymi, udało się osiągnąć bardzo ciekawe rezultaty.

b. Doświadczenia po wdrożeniu

Niewątpliwie moje doświadczenia okazały się być bardzo pozytywne. Złożyło się na nie kilka kluczowych kwestii:

- prostota użycia Neo4j oraz Cypher
- przystępnie napisana dokumentacja technologii wraz z bogatą bazą poradników wdrożeniowych
- naturalność rozwiązania - baza danych nie wymaga wysokiego poziomu abstrakcji tworzonych rozwiązań; dla twórcy programowanie wydaje się być naturalne i logiczne
- wyjątkowo dopracowana warstwa graficzna oprogramoania na desktop i przeglądarki, która umożliwia wygodne tworzenie i testowanie kodu oraz dostęp do danych

Pomimo wielu dobrych doświadczeń doświadczyłem także kilku problemów:

- istnieje dość niewielka baza wiedzy nt. rozwiązania poza główną dokumentacją, co utrudnia poszukiwanie odpowiedzi na problemy poza murami serwisu
- zastosowana składnia języka Cypher często wymusza tworzenie długich zapytań, których głównym elementem jest ta sama procedura przekazywania zmiennych do kolejnych procedur; możliwość tworzenia stałych zmiennych, dostępnych z każdego miejsca w zapytaniu ułatwiłaby i wyjątkowo skróciła dłuższe formuły

c. Potencjał dalszego rozwoju

Zdecydowałem się opisać potencjał dalszego rozwoju z uwagi na zaskakującą jak dla mnie możliwość analizy danych, choćby na podstawie kilku poglądowych metod. Dla nieskomplikowanej bazy danych, która posłużyła za przykład dla tego projektu potencjał generowania relacji i rozwijania bazy jako platformy jest ogromny. Wielokrotnie decydowałem się na okrojenie rozwiązania, z uwagi na możliwość dynamicznego rozwoju na wiele sposobów.
Kilka możliwych dróg rozwoju:

- możliwość dodania większej ilości danych do uzyskania lepszych, bardziej statystycznych wyników
- możliwość dodania nowych typów relacji oraz nowych rodzajów węzłów
- więcej możliwości analizy zebranych danych dzięki szeregowi przystępnym algorytmów w zakresie data science