SMART-ROUTE

projekt tras transportu publicznego dla użytkowników

Tworzony przez społeczność: NEXT LEVEL DEV

AGENDA:

- ***Cel - podjęcie decyzji co do wyboru odpowiedniej bazy danych***

- przedstawienie funkcjonalności aplikacji
- przedstawienie wady/zalety popularnych baz danych sql | nosql
- jakie bazy danych obecnie używa się w aplikacjach komercyjnych
- jakie funkcjonalności w/w bazy oferują
- która baza danych najlepiej spełnia wymagania aplikacji smart-route pod względem:
    - funkcjonalności którą oferuje
    - łatwości implementacji
    - szybkości działania
    - obciążenia serwera

1. Funkcjonalności aplikacji:

- Historia wyszukiwania tras przez użytkownika.
- Wybór trasy z historii wyszukiwania i ponowne jej wyszukanie.
- Wybór trasy z historii wyszukiwania i jej cofnięcie.
- Wyświetlanie najczęściej wyszukiwanej trasy z ostatnich 24 godzin wśród wyszukiwań w całej aplikacji.
- Obliczanie czasu dla wyszukiwanej trasy.
- Wyszukiwanie trasy z możliwością zatrzymania się w wybranych miejscach po drodze.
- Zaplanowanie trasy składającej się z n punktów i dodanie czasu, jaki spędzę w każdym z nich, pozwoli aplikacji obliczyć, ile czasu zajmie cała podróż.
- Zmieniając czas, jaki spędzę w danym punkcie, aplikacja przeliczy dla mnie, ile czasu zajmie cała podróż i z których połączeń powinienem skorzystać.
- Planowanie wycieczek do parków, muzeów lub innych miejsc. Chciałbym, aby aplikacja pokazała mi, jak się między nimi poruszać.
- Planowanie trasy między n punktami, biorąc pod uwagę fakt, że chcę przejść pieszo między m z nich.
- Podstawowe wyszukiwanie trasy (początek, miejsce docelowe, optymalizacja czasu).
- Historia wyszukiwania + możliwość ponownego wyszukiwania tras.
- Odwrócenie trasy (z A → B do B → A).
- Szacowany czas podróży (na podstawie danych komunikacyjnych).

Wymagania systemowe

- Wymagania funkcjonalne

- Obsługa danych geolokalizacyjnych - dane z API
- Historia wyszukiwań i operacje transakcyjne
- Dynamiczne obliczenia tras
- Wysoka dostępność oraz niskie czasy odpowiedzi

- Wymagania niefunkcjonalne

- Skalowalność
- Spójność i integralność danych
- Łatwość implementacji i utrzymania

2. Charakterystyka baz danych SQL | NoSQL – wady i zalety

Bazy SQL

Zalety

1. **Model danych i transakcyjność**
   Struktura relacyjna - dane są przechowywane w tabelach o z góry określonym schemacie, z ustalonymi relacjami (klucze główne/obce).
   Umożliwia to tworzenie złożonych zapytań, co jest istotne przy analizie historii wyszukiwań

2. **Wsparcie dla danych geolokalizacyjnych**
   Rozszerzenie PostGIS w PostgreSQL - umożliwia przechowywanie i przetwarzanie danych przestrzennych, co jest kluczowe przy integracji z API Warszawskiego Transportu Publicznego

Wady

3. **Skalowalność** - skalowanie pionowe (dodanie mocy do jednego serwera) może być ograniczeniem przy ekstremalnych obciążeniach
4. **Sztywność schematu** - wymaga z góry zdefiniowanego schematu, co może ograniczać elastyczność przy szybko zmieniających się wymaganiach

Bazy NoSQL

Zalety

1. **Elastyczność schematu:** Pozwala na przechowywanie danych o zmiennej strukturze – przydatne, gdy dane (np. logi z API) nie są ściśle ustrukturyzowane
2. **Skalowalność**: Umożliwia dodawanie kolejnych serwerów, co jest korzystne przy rosnącej liczbie operacji w czasie rzeczywistym.
3. **Wysoka wydajność przy dużych wolumenach danych:** NoSQL często lepiej radzi sobie z szybkim zapisem i odczytem danych.

Wady

1. **Brak pełnych gwarancji ACID**: Może to utrudniać obsługę transakcji, szczególnie przy operacjach, gdzie spójność danych jest krytyczna (np. historia wyszukiwania tras).
2. **Mniej zaawansowane zapytania**: Operacje złożonych zapytań, łączenia danych i transakcji mogą być trudniejsze do implementacji.
3. **Potencjalne problemy z duplikacją danych**: Brak relacyjności oznacza, że trzeba samodzielnie dbać o integralność relacji między danymi.

Obecnie wykorzystywane bazy danych w aplikacjach:

SQL: PostgreSQL, MySQL, Oracle, Microsoft SQL Server

NoSQL MongoDB, Cassandra, Redis, Couchbase

Wstępne porównanie SQL i NoSQL w kontekście aplikacji SMART-ROUTE

1. **Model danych i transakcyjność**

- **SQL**:
  – Stały schemat, co zapewnia uporządkowanie danych (np. historia wyszukiwań, informacje o trasach).
  – Pełne wsparcie transakcji (ACID) gwarantujące spójność, co jest krytyczne przy operacjach związanych z planowaniem podróży.
- **NoSQL**:
  – Elastyczność modelu (przydatna przy przechowywaniu nieustrukturyzowanych danych z API), ale brak pełnych transakcji może stanowić ryzyko przy operacjach wymagających wysokiej spójności.

2. **Skalowalność**

- **SQL** (np. PostgreSQL):
  – Skalowanie pionowe może być wystarczające przy umiarkowanym wzroście, a dostępne są rozwiązania klastrowe.
  – Dobre narzędzia optymalizacyjne dla zapytań przestrzennych dzięki rozszerzeniom PostGIS.
- NoSQL:
  – Skalowanie poziome idealnie sprawdzi się w sytuacjach, gdy liczba operacji odczytu/zapisu gwałtownie rośnie (np. dane w czasie rzeczywistym z pojazdów).
  – Może być lepszym wyborem w aplikacjach big data, jednak w naszym przypadku kluczowa jest spójność transakcyjna.

3. **Wsparcie dla operacji geolokalizacyjnych**

- **SQL:**
  – PostgreSQL z PostGIS oferuje funkcje przetwarzania danych geolokalizacyjnych, co pozwala na szybkie obliczenia odległości, wyszukiwanie w obrębie promienia.
- **NoSQL:**
  – Niektóre bazy dokumentowe (np. MongoDB) oferują wsparcie dla zapytań geospatial, ale często nie są tak rozbudowane jak w przypadku dedykowanych rozszerzeń SQL.

4. **Wydajność zapytań i integracja z API**

- **SQL:**
  – Język SQL umożliwia tworzenie bardzo precyzyjnych zapytań – istotnych przy obliczeniach tras i analizie historii.
  – Możliwość optymalizacji zapytań poprzez indeksy i materializowane widoki.

- **NoSQL:**
  – Szybki zapis i odczyt, co jest korzystne przy wysokiej częstotliwości operacji, ale skomplikowane zapytania (np. agregacje) mogą wymagać dodatkowej logiki w aplikacji

5. **Koszty operacyjne i zarządzanie**

- **SQL:**
  – Utrzymanie bazy o stałym schemacie może wymagać większych nakładów przy dużych obciążeniach, ale stabilność i dojrzałość rozwiązań są kluczowe dla systemów krytycznych.

- **NoSQL:**
  – Niższe koszty skalowania poziomego i elastyczność mogą być atrakcyjne, jednak potencjalne problemy ze spójnością danych mogą zwiększyć koszty wdrożenia zabezpieczeń i logiki aplikacyjnej.