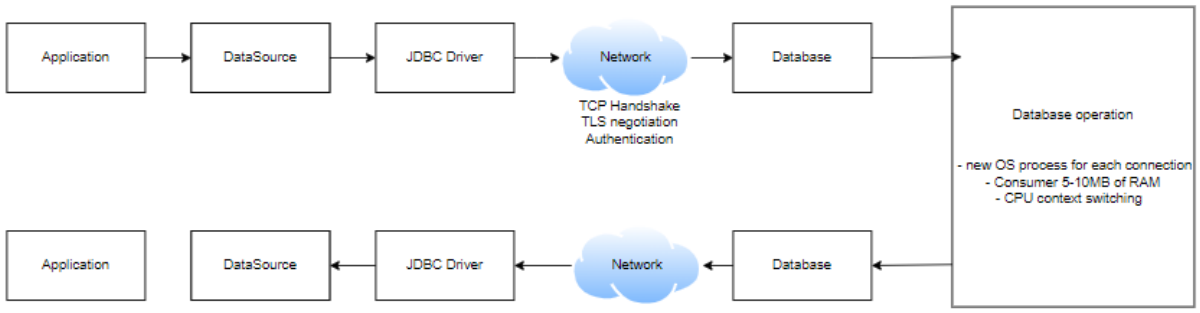
JPA Optimalizations

Najczęstsze przyczyny wolnego działania JPA

1. Za duża liczba połączeń do bazy danych
2. Utworzone zapytania są zbyt wolne (nie zoptymalizowane)
3. Błędne mapowania JPA
4. Pobieranie większej ilości danych niż potrzeba

Wykorzystywanie Connection Pool



W celach optymalizacyjnych stosuje się mechanizm Connection Pool. Jednym z najlepszych dostawców CP jest HikariCP (https://github.com/brettwooldridge/HikariCP).

Podstawowe parametry:

1. autoCommit - oznacza, że każda pojedyncza operacja na bazie danych będzie traktowana jako osobna transakcja- czyli może może zwrócić commit, albo rollback.
2. connectionTimeout - maksymalny czas oczekiwania na połączenie z puli
3. idleTimeout - czas bezczynności połączeń w puli, po której następuje jej usunięcie. Minimalna wartość to 10 sekund.

4. keepaliveTime - parametr, który określa po jakim czasie hikari wysyła zapytanie typu “keepalive”, które podtrzymuje połączenie z bazą danych. Ma to na celu utrzymania połączenia aktywnego, i zapobiegnięcie przed zakończeniem połączenia przez sieć lub bazę danych
5. maxLifetime - określa czas po jakim połączenie (nawet aktywne) zostaje usunięte. Zapobiega to powstawaniu długotrwałych połączeń, które mogą stać się niestabilne.
6. connectionTestQuery (jeżeli driver wspier **JDBC4** nie należy tego parametru ustawiać - np postgres) - sprawdza poprawność połączenia
7. minimumIdle - minimalna liczba połączeń bezczynnych, jaką HikariCP próbuje ustawiać. Zalecane nie jest ustawianie tej wartości, a zamiast tego rekomendowane jest ustawienie HikariCP na stałą wartość połączeń używając maximumPoolSize
8. maximumPoolSize (ważny parametr) - parametr, który określa, jaka jest maksymalna ilość połączeń (łącznie aktywnych i nieaktywnych) ten parametr najlepiej dobrać bazując na danych z środowiska działania, oraz dokumentacji odnośnie wybrania odpowiedniej wartości: https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing. Default value = 10. Pula ma być mała (najlepiej maksymalnie z kilkudziesięcioma połączeniami), a reszta wątkow ma być zablokowana na puli, oczekując na połączenia.

Narzędzia do optymalizacji czasu połączeń

1. FlexyPool - https://github.com/vladmihalcea/flexy-pool
2. DatasourceProxy - https://github.com/jdbc-observations/datasource-proxy
3. Możemy zaimplementować oba narzędzia poprzez Spring **Boot DataSource Decorator -** https://github.com/gavlyukovskiy/spring-boot-data-source-decorator?tab=readme-ov-file
3. DIGMA - DIGMA to platforma ciągłego feedbacku, która identyfikuje i rozwiązuje problemy wydajnościowe w kodzie. Integruje się z środowiskami deweloperskimi, automatycznie wskazując wąskie gardła i problemy ze skalowalnością, oferując wgląd w czasie rzeczywistym bez opuszczania IDE.

DIGMA wykorzystuje OpenTelemetry do instrumentacji i przeprowadza analizy lokalnie, nie wymagając zmian w kodzie ani przesyłania danych do chmury. Jest bezpłatna do użytku lokalnego, co zwiększa wydajność deweloperów.

Use case

Case 1

1. SpringJPA open in view - ta wartość jest domyślnie włączona. Powoduje ona, to, że w przypadku, gdy chcemy utworzyć encje, które nie są pełne, z powodu pola na inny obiekt który jest ustawiony na @LazyInitialization. W niektórych przypadkach (np. jak poniżej powoduje, że mimo, że tylko 1 metoda kontrolera jest @Transactional, to na całe wywołanie jest połączenie aktywne.

```
    //SampleService
    @Transactional
    public void hello() {
        System.out.println(personRepository.findAll());
    }

   //SampleController
       @GetMapping("/hello")
            void hello() {
        sampleService.hello();
        externalService.externalCall(); //Sleep.sleep(200);
    }
```

2. Rozwiązanie → w pliku konfiguracyjnym ustawiamy: spring.jpa.open.in-view=false. Ustawienie to mówi, żeby nastąpiła generacja widoku z encjami niepełnymi (lazyLoading)

Case 2

```
    //SampleService
```

```
    @Transactional
    public void withExternalServiceCall() {
        externalService.externalCall();
        System.out.println(personRepository.findAll());
    }

    @Transactional
    public void withExternalServiceCallAfter() {
        System.out.println(personRepository.findAll());
        externalService.externalCall();
    }
```

W tym przypadku, chcemy, żeby tylko pobieranie danych wykorzystywało połączenie z bazą danych, a reszta kodu nie.

Możliwe rozwiązania:

1. AutoCommit (spring.datasource.hikari.auto-commit=false) - Jednak to rozwiązanie działa tak w przypadku, gdy jest oko ostatnim wywołaniem w metodzie -w poniższym przykładzie nie zadziała, bo pierwsze jest wywołanie metody z połączeniem do bazy danych.

Idzie to rozwiązać stosując TransactionTemplate w którym definujemy, jaka część jest tranzakcyjna.

```
    private final TransactionTemplate transactionTemplate;

    @Transactional
    public void withExternalServiceCallAfter() {
        transactionTemplate.executeWithoutResult(
                transactionStatus -> {
                    System.out.println(personRepository.findAll()
                }
        );
```

```
        externalService.externalCall();
    }
```

Case 3

```
    public void execute(String bankTransferId,
                        String reference,
                        String senderId,
                        String receiverId,
                        Amount amount) {
        Account sender = accountRepository.findByIdOrThrow(sende
        Account receiver = accountRepository.findByIdOrThrow(rec

        BankTransfer bankTransfer = new BankTransfer(bankTransfe
        bankTransferRepository.save(bankTransfer);
    }
```

Co jest tutaj źle?

1. BankTransfer posiada pole @Id private String id - i wywołując save na repozytorium wchodzi on w SimpleJpaRepository jest sprawdzenie, czy jest to nowa encja czy nie.

```
    @Transactional
     public <S extends T> S save(S entity) {
        Assert.notNull(entity, "Entity must not be null");
        if (this.entityInformation.isNew(entity)) {
            this.entityManager.persist(entity);
            return entity;
        } else {
            return this.entityManager.merge(entity);
        }
    }
```

1. Ponieważ ma pole Id, oznacza to, że nie jest to nowa encja i wywoływane jest .merge, który potrzebuje połączenia do bazy danych w celu pobrania potrzebnych danych.

   Rozwiązaniem jest powiedzenie Spring, że to jest nowa encja - robimy to zamieniają dodając @Version - powieważ on jest jako null domyślnie → nowy obiekt

   ```
   @Version
   private Long version;
   //or
   public class BankTransfer implements Persistible {

   }
   ```

2. Nalezy również dodać annotację @Transactional

Ostatnią optymalizacją w tym przypadku jest pobranie referencji poprzez Id, w celu utworzenia nowego transferu bankowego. Robimy to poprzez metodę getReferenceId(objId);

```
    public void execute(String bankTransferId,
                        String reference,
                        String senderId,
                        String receiverId,
                        Amount amount) {
        Account sender = accountRepository.getReferenceId(sender
        Account receiver = accountRepository.getReferenceId(rece

        BankTransfer bankTransfer = new BankTransfer(bankTransfe
        bankTransferRepository.save(bankTransfer);
    }
```

Wadą tego jest brak sprawdzenia, czy obiekty składowe (sender, receiver) istnieją w bazie danych i możemy uzyskać ConstraintViolationException exception - podsumowanie, jeżeli chcesz modyfikować obiekty to je pobieraj, natomiast do wyświetlania używaj projekcji.

Case 4

Jeżeli w encji chcemy zmienić pojedyncze pole, np.

```
    public void execute(String bankTransferId) {
        BankTransfer bankTransfer = bankTransferRepository.findB
        bankTransfer.settle();
        bankTransferRepository.save(bankTransfer);
    }
```

i klasa ma połączenia np. ManyToOne z Account, to domyślnym parametrem feach jest eager, co powoduje niepotrzebne pobieranie w tym przypadku tycvh danych.

Rozwiązaniem jest zmiana fetch na lazy

Case 5 - N+1

```
@Transactional
    public void execute(String senderId) {
        List<BankTransfer> entries = bankTransferRepository.find

        for (BankTransfer sentTransfer : entries) {
            System.out.println(sentTransfer.getReceiver().getIba
        }
    }
```

Problem n+1 polega na tym, że dla pojedynczego zapytania, dla każdego rezultatu wykonujemy ponownie zapytanie.

Rozwiązaniem jest ustawienie feach na lazy dla encji pobieranych, oraz w repozytorium JPA wpisać ręcznie named Query

```
@Query(from BankTransfer bank join fetch bank.sender join fetch
    List<BankTransfer> findBySenderId(String senderId);
```

Mniej optymalniejszym rozwiązaniem jest zamiast named query, użycie @EntityGraph.

```
@EntityGraph(attributePaths = {"sender","receiver"})
List<BankTransfer> findBySenderId(String senderId);
```

Po naprawie wykonuje się tylko jedno zapytanie select i jedno zapytanie update.

Dalej jednak nie jest to optymalne, ponieważ update wykonuje operacje na wszystkich kolumnach. Aby to zmienić, należy dodać annotacje @DynamicUpdate

Narzędzie

1. https://github.com/vladmihalcea/hypersistence-optimizer - commercial
2. https://github.com/quick-perf/quickperf - free to use

Qucikperf - narzędzie do testowania oczekiwanej ilości zapytań z uwzględnieniem rodzaju operacji

Działa poprzez dodawanie annotacji:

```
@ExpectSelect()
@ExpectUpdate()
@ExpectDelete()
```

Projekcje

Posiadając określoną klasę, możemy utworzyć klasę projekcyjną która będzie posiadała pola odpowiadające naszej klasie bazowej. Następnie możemy w repozytorium dodać daną projekcje, która będzie służyła, tylko do wyświetlania.

```
@Entity
public class Account {
    @Id
    private String id;
    private String iban;
    private String firstName;
```

```
    private String lastName;

    @ElementCollection
    @CollectionTable(
        name = "phone_number",
        joinColumns = @JoinColumn(name = "account_id")
    )
    private List<PhoneNumber> phoneNumbers;
}

record NamesOnly(String id, String firstName, String lastName) {

public interface AccountRepository extends JpaRepository<Account
    NamesOnly findNamesOnlyById(String id);
}
```

W przypadku niedopasowania nazw pól, możemy użyć named query

Wadą tego rozwiązania jest potrzeba tworzenia dużej ilości projekcji, dlatego można zastosować mechanizm zwany dynamic projections:

```
public interface AccountRepository extends JpaRepository<Account
    <T> T findById(String id, Class<T> clazz);
}
```

Powyższy dokument powstał na podstawie prezentacji - Performance oriented Spring Data JPA & Hibernate by Maciej Walkowiak