Advanced Query Techniques and Performance Optimization

The hospital management system built with Spring Boot leverages various advanced techniques for managing complex data and optimising performance. This document provides an in-depth look at the usage of complex queries, dynamic query building, query derivation, pagination and sorting, fetching strategies, and performance tuning within the system.

1. Complex Queries Using JPQL and Native SQL

The project uses both JPQL and native SQL queries to handle complex data retrieval scenarios involving multiple joins and conditional statements. This approach is evident in the `EmployeeRepository`, where a native SQL query is used to aggregate data across several related tables:

```
@Query(value = "SELECT e.id, e.surname, e.first_name, e.address, e.telephone_number," +
        " d.speciality, n.salary, n.rotation, d2.name as departmentName, d3.\"name\" as
departmentSupervise" +
        " FROM employee e" +
        " LEFT JOIN doctor d ON e.id = d.id" +
        " LEFT JOIN nurse n ON e.id = n.id" +
        " LEFT JOIN department d2 on n.department_id = d2.id" +
        " LEFT JOIN department d3 on d.id = d3.director_id",
        nativeQuery = true)
Page<EmployeeProjection> findAllEmployees(Pageable pageable);
```

This query efficiently pulls together employee details along with specific attributes from the `Doctor` and `Nurse` subclasses, demonstrating the system's ability to execute highly complex queries efficiently.

2. Specifications for Dynamic Query Building

Specifications allow for the construction of dynamic queries based on application runtime requirements. In the `EmployeeSpecification` class, methods are designed to return `Specification<Employee>` objects that are used to build queries conditionally:

```
public class EmployeeSpecification {
    public static Specification<Employee> hasFirstName(String firstName) {
        return (root, query, cb) -> cb.equal(root.get("firstName"), firstName);
    }

    public static Specification<Employee> hasSurname(String surname) {
        return (root, query, cb) -> cb.equal(root.get("surname"), surname);
    }
}
```

These specifications are utilised in the service layer to filter employees dynamically based on user-provided criteria:

```
public List<Employee> findEmployees(String firstName, String surname) {

    return employeeRepository.findAll(Specification
            .where(EmployeeSpecification.hasFirstName(firstName))
            .or(EmployeeSpecification.hasSurname(surname)));
}
```

3. Query Derivation

Spring Data simplifies the implementation of data access operations by automatically deriving SQL queries from repository method signatures. This feature is used extensively in the project to implement basic data retrieval operations without writing explicit SQL or JPQL queries:

```
List<Employee> findByAddress(String address);
```

4. Pagination and Sorting

To manage large datasets effectively, the system implements pagination and sorting, which are crucial for enhancing performance and improving user experience. This is demonstrated in the `EmployeeService` implementation:

```
public Page<EmployeeProjection> getAllEmployees(int page, int size) {
    Pageable pageable = PageRequest.of(page, size, Sort.by("id").ascending());
    return employeeRepository.findAllEmployees(pageable);
}
```

5. Fetching Strategies

The project uses various fetching strategies to optimise data loading. Lazy loading is implemented in the `Nurse` entity to prevent unnecessary data retrieval, enhancing performance:

```
@JsonIgnore
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "department_id")
private Department department;
```

6. Performance Tuning

Performance optimizations include the use of caching and custom transaction annotations to reduce load times and improve transaction efficiency:

Caching Example:

```
@Cacheable(value = "doctorsBySpeciality", key = "#speciality", unless = "#result.isEmpty()")
public List<Doctor> findDoctorsBySpeciality(String speciality) {
    return doctorRepository.findBySpeciality(speciality);
}
```

This comprehensive use of advanced querying and performance optimization techniques ensures that the hospital management system is capable of handling complex operations efficiently, thereby providing robustness and scalability.