Summary of Advanced Query Techniques and Performance Optimization

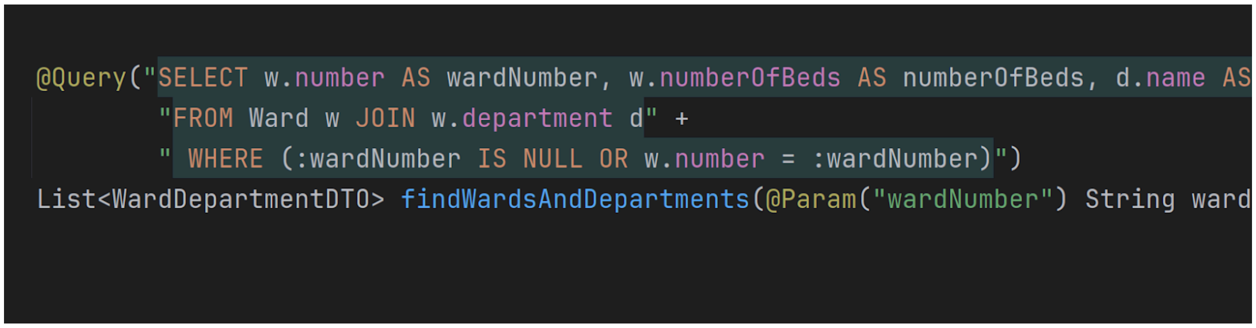
As I've been diving deeper into Spring Data JPA, I've realized the importance of crafting efficient queries that not only retrieve the necessary data but also do so in an optimal way. This has led me to explore various advanced query techniques and how they can be optimized for performance in this project.

1. Leveraging JPQL and Native Queries

In my Spring Data JPA project, I've primarily used two types of queries: JPQL and Native Queries.

JPQL

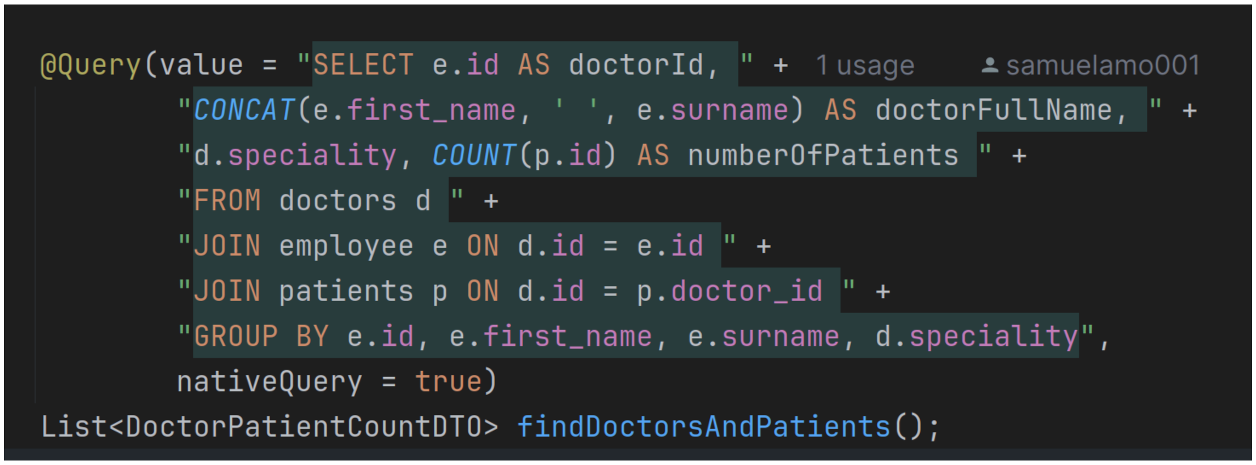
JPQL (Java Persistence Query Language) is the go-to for most of my database interactions. It's powerful, expressive, and integrates seamlessly with JPA entities. For instance, to retrieve all departments along with the number of wards they have, I wrote:



```
@Query("SELECT w.number AS wardNumber, w.numberOfBeds AS numberOfBeds, d.name AS
        "FROM Ward w JOIN w.department d" +
        " WHERE (:wardNumber IS NULL OR w.number = :wardNumber)")
List<WardDepartmentDTO> findWardsAndDepartments(@Param("wardNumber") String ward
```

Native Queries

While JPQL works great in most scenarios, there are times when I needed the raw power of SQL. Native Queries have been my tool of choice in such cases. For example, to fetch doctors and the count of their patients, I used:



```
@Query(value = "SELECT e.id AS doctorId, " +  1 usage   samuelamo001
        "CONCAT(e.first_name, ' ', e.surname) AS doctorFullName, " +
        "d.speciality, COUNT(p.id) AS numberOfPatients " +
        "FROM doctors d " +
        "JOIN employee e ON d.id = e.id " +
        "JOIN patients p ON d.id = p.doctor_id " +
        "GROUP BY e.id, e.first_name, e.surname, d.speciality",
        nativeQuery = true)
List<DoctorPatientCountDTO> findDoctorsAndPatients();
```

2. Optimizing Query Performance

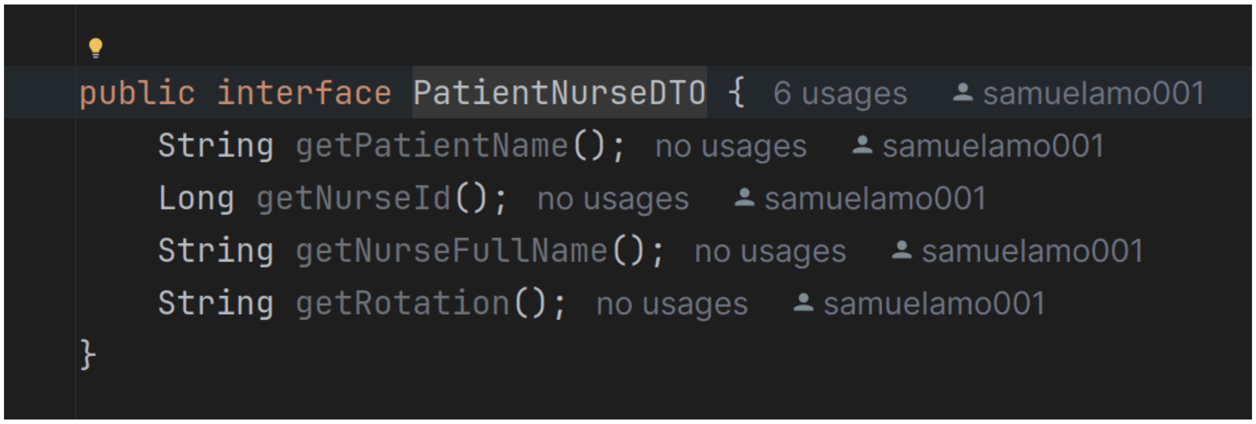
Here's what I've learned and implemented about optimizing query performance

Avoiding N+1 Selects

N+1 select problem is something I encountered early on. To avoid this, I started using `@Query` in JPQL to ensure that related entities are fetched in a single query rather than multiple queries.

Using Projections

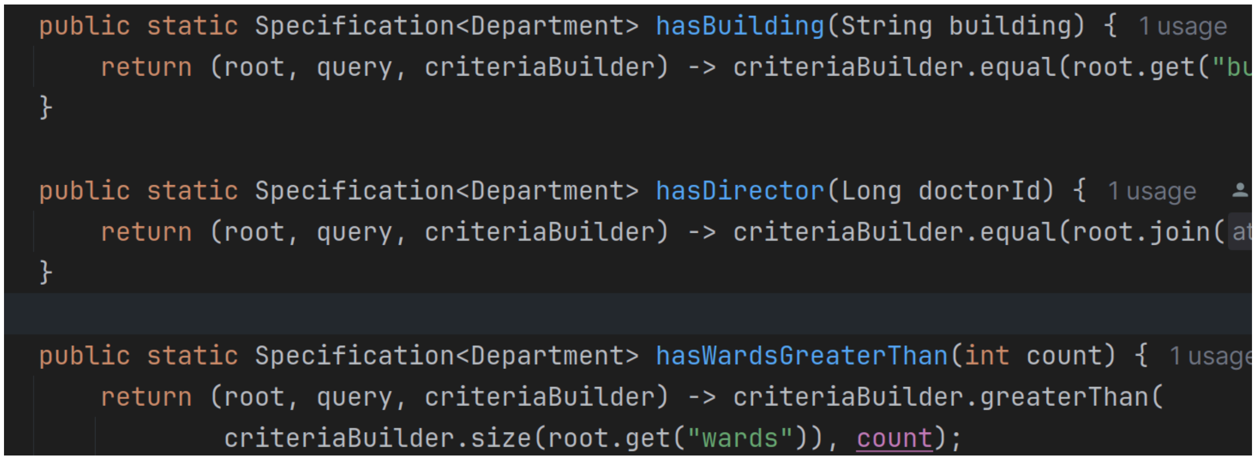
To improve performance, especially in read-heavy operations, I utilized projections. Instead of fetching entire entities, I only retrieved the specific fields I needed. DTO-based projections, like the one below, significantly reduced the amount of data retrieved:



```
public interface PatientNurseDTO {  6 usages   samuelamo001
    String getPatientName();  no usages   samuelamo001
    Long getNurseId();  no usages   samuelamo001
    String getNurseFullName();  no usages   samuelamo001
    String getRotation();  no usages   samuelamo001
}
```

3. Specifications for Dynamic Queries

Dynamic queries are a powerful feature of Spring Data JPA, and I've used specifications to build them. This approach has allowed me to construct queries at runtime based on user input or other conditions. For example, to find all departments with more than a certain number of wards, I wrote:



```
public static Specification<Department> hasBuilding(String building) { 1 usage
    return (root, query, criteriaBuilder) -> criteriaBuilder.equal(root.get("bu
}

public static Specification<Department> hasDirector(Long doctorId) { 1 usage
    return (root, query, criteriaBuilder) -> criteriaBuilder.equal(root.join( at
}

public static Specification<Department> hasWardsGreaterThan(int count) { 1 usage
    return (root, query, criteriaBuilder) -> criteriaBuilder.greaterThan(
            criteriaBuilder.size(root.get("wards")), count);
```

Conclusion

The journey of mastering advanced query techniques and performance optimization in Spring Data JPA has been rewarding. By carefully selecting the right type of query, optimizing them for performance, and leveraging the power of specifications, I've been able to build a robust and efficient data access layer for this application application.