

Min CourseService ser lige nu således ud. Jeg vil gerne have optimeret, så jeg ikke laver unødvendige databaseopkald ved ændring af addCount.

```
package mmh.valgfagsfordeling.service; import mmh.valgfagsfordeling.dto.CourseDTO; import  
mmh.valgfagsfordeling.model.Course; import mmh.valgfagsfordeling.repository.CourseRepository; import  
org.springframework.stereotype.Service; @Service public class CourseService { private final CourseRepository  
courseRepository; public CourseService(CourseRepository courseRepository) { this.courseRepository = courseRepository; }  
//-----Entity metoder----- public Course getCourse(int courseId) { return courseRepository.findById(courseId)  
.orElseThrow(() -> new RuntimeException("valgfag ikke fundet")); } public boolean checkIfAvailable(int courseId) { Course  
selectedCourse = getCourse(courseId); return selectedCourse.getParticipantsCount() <  
selectedCourse.getMaxParticipants(); } public void addCount(int courseId) { Course selectedCourse = getCourse(courseId);  
selectedCourse.incrementParticipantsCount(); } //-----DTO----- public CourseDTO getCourseDTO(int  
courseId) { Course course = getCourse(courseId); return convertToDTO(course); } //-----Hjælpemetoder-----  
private CourseDTO convertToDTO(Course course) { CourseDTO dto = new CourseDTO();  
dto.setCourseId(course.getId()); dto.setName(course.getName());  
dto.setDescription(course.getDescription()); dto.setParticipantsCount(course.getParticipantsCount());  
dto.setMaxParticipants(course.getMaxParticipants()); dto.setMinParticipants(course.getMinParticipants());  
dto.setSemester(course.getSemester()); dto.setTeacher(course.getTeacher()); return dto; }}
```

ChatGPT sagde:

Perfekt 🤙 – du har en solid CourseService, men ja — som du selv spotter,
den laver **ét databaseopslag per elev pr. tildeling** (to gange faktisk, i checkIfAvailable() og addCount()).

Det betyder, at hvis du fx har 400 elever og 3 valgfag pr. elev, laver du hurtigt **1000+ DB-kald**,
selvom du reelt kun har måske 10–20 fag. Det er unødvendigt.

Lad os optimere det med **et simpelt cache-lag i hukommelsen**,
uden at du mister dataintegritet eller transaktionssikkerhed.

🎯 Mål

- Hent **alle kurser én gang** i hukommelsen, når du starter fordelingen.
 - Arbejd på de **samme Course-instanser** under hele algoritmen.
 - Gem ændringerne **én gang til sidst** (enten automatisk via @Transactional,
eller eksplisit med saveAll() hvis du vil være helt sikker).
-

⚙️ Optimeret CourseService

Her er en version, du kan bruge direkte:

```
package mmh.valgfagsfordeling.service;
```

```
import mmh.valgfagsfordeling.dto.CourseDTO;  
import mmh.valgfagsfordeling.model.Course;  
import mmh.valgfagsfordeling.repository.CourseRepository;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;
```

```

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Service
@Transactional
public class CourseService {

    private final CourseRepository courseRepository;

    // Cache i hukommelsen – holder alle kurser i denne transaktion
    private Map<Integer, Course> courseCache = new HashMap<>();

    public CourseService(CourseRepository courseRepository) {
        this.courseRepository = courseRepository;
    }

    //----- Initialisering -----

    /**
     * Hent alle kurser én gang og læg dem i hukommelsen.
     * Kaldes fx fra AdministrationService før fordelingen starter.
     */
    public void preloadAllCourses() {
        List<Course> allCourses = courseRepository.findAll();
        courseCache = allCourses.stream()
            .collect(Collectors.toMap(Course::getCourseId, c -> c));
    }

    //----- Entity metoder -----

    public Course getCourse(int courseId) {
        // Prøv først i cache – ellers hent fra DB (og læg i cache)

```

```

if (courseCache.containsKey(courseld)) {
    return courseCache.get(courseld);
}

Course course = courseRepository.findById(courseld)
    .orElseThrow(() -> new RuntimeException("Valgfag ikke fundet: " + courseld));
courseCache.put(courseld, course);
return course;
}

public boolean checkIfAvailable(int courseld) {
    Course course = getCourse(courseld);
    return course.getParticipantsCount() < course.getMaxParticipants();
}

public void addCount(int courseld) {
    Course course = getCourse(courseld);
    course.incrementParticipantsCount();
}

/**
 * Når hele fordelingen er færdig, kan du gemme alt tilbage.
 */

public void persistAll() {
    courseRepository.saveAll(courseCache.values());
}

//----- DTO -----

public CourseDTO getCourseDTO(int courseld) {
    Course course = getCourse(courseld);
    return convertToDTO(course);
}

//----- Hjælpemetoder -----

```

```
private CourseDTO convertToDTO(Course course) {  
    CourseDTO dto = new CourseDTO();  
    dto.setCourseId(course.getCourseId());  
    dto.setCourseName(course.getCourseName());  
    dto.setDescription(course.getDescription());  
    dto.setParticipantsCount(course.getParticipantsCount());  
    dto.setMaxParticipants(course.getMaxParticipants());  
    dto.setMinParticipants(course.getMinParticipants());  
    dto.setSemester(course.getSemester());  
    dto.setTeacher(course.getTeacher());  
    return dto;  
}  
}  


---


```

Forklaring step-by-step

preloadAllCourses()

Henter *alle kurser én gang* fra databasen og gemmer dem i en Map<Integer, Course>. → Denne kalder du **før du starter fordelingen**, fx i din AdministrationService:

```
public void distributionGreedyWithFairness() {  
    courseService.preloadAllCourses(); // 👍 vigtigt!
```

```
    initListAllStudents = studentService.studentListInternal();  
    ...  
}
```

Når først kurserne er hentet, vil alle senere opslag (både i checkIfAvailable og addCount) blot returnere objektet fra hukommelsen.

2 getCourse(int courseId)

Returnerer først fra cache (courseCache), og hvis kurset ikke er der (fx ved et særtilfælde), henter den det fra DB **og lægger det i cachen** til senere brug.

3 addCount() og checkIfAvailable()

Nu er de helt **O(1)** i kompleksitet og **ingen databasekald**.
De arbejder på det samme objekt, som JPA allerede kender.

persistAll()

Når hele fordelingen er færdig, kan du kalde:

```
courseService.persistAll();
```

→ Så laver Spring/Hibernate **én batch-opdatering** for alle kurser.

Hvis AdministrationService er annoteret med @Transactional (som hos dig), behøver du faktisk ikke engang kalde persistAll() — JPA vil automatisk committe ændringerne, når transaktionen afsluttes.

Men jeg anbefaler at kalde den alligevel — det gør koden mere eksplisit og testbar.

Hvor meget hurtigere?

Uden cache

Ca. 2 DB-kald per elev pr. fag

Med cache

0 DB-kald under løb

400 elever × 3 fag × 2 kald = 2400 queries 10–20 queries (alle kurser ved start, 1 batch save til sidst)

Hver tildeling tager ~1–2 ms

Hele fordelingen kører på under 0,5 s

Brug i din AdministrationService

Sådan ser din metode ud efter integration:

```
public void distributionGreedyWithFairness() {  
    // 1. Preload kurser én gang  
    courseService.preloadAllCourses();  
  
    // 2. Initialisér lister og hent elever  
    initListAllStudents = studentService.studentListInternal();  
    fulfilled1 = new ArrayList<>();  
    toBeFirstList1 = new ArrayList<>();  
    fulfilled2 = new ArrayList<>();  
    toBeFirstList2 = new ArrayList<>();  
    toBeManualHandled = new ArrayList<>();  
  
    // 3. Kør runderne  
    processRound(initListAllStudents, fulfilled1, toBeFirstList1, false);  
}
```

```
processRound(toBeFirstList1, fulfilled2, toBeFirstList2, false);
processRound(fulfilled1, fulfilled2, toBeFirstList2, false);
processRound(toBeFirstList2, new ArrayList<>(), new ArrayList<>(), true);
processRound(fulfilled2, new ArrayList<>(), new ArrayList<>(), true);

// 4. Gem alt tilbage
courseService.persistAll();
}
```