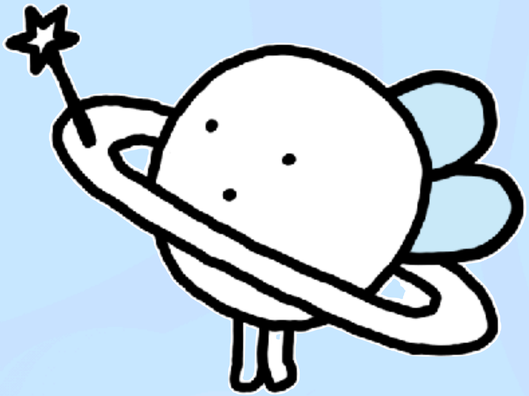
KBO 크롤러
성능 및 안정성 개선

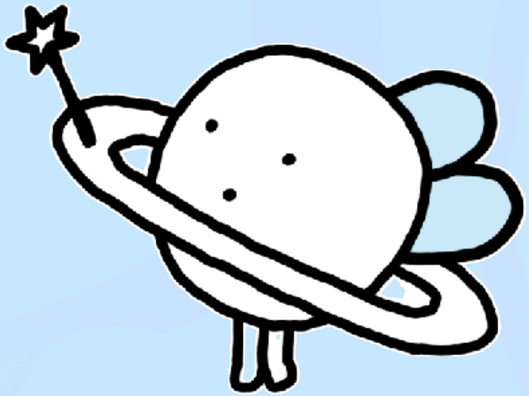
BE 7기 밍트

개요



1. 성능 계측
2. 크롤러 성능 개선
3. 운영 안정성 개선
4. 마무리

개요



1. 성능 계측
2. 크롤러 성능 개선
3. 운영 안정성 개선
4. 마무리

1. 성능 계측

경기 결과 조회 크롤링 성능 측정

상태별 폴링 간격

상태	경과 시간	폴링 간격	비고
SCHEDULED	< 120분	10분	일반 대기 상태
SCHEDULED	≥ 120분	15분	장기 지연 시
LIVE	-	5분	경기 진행 중

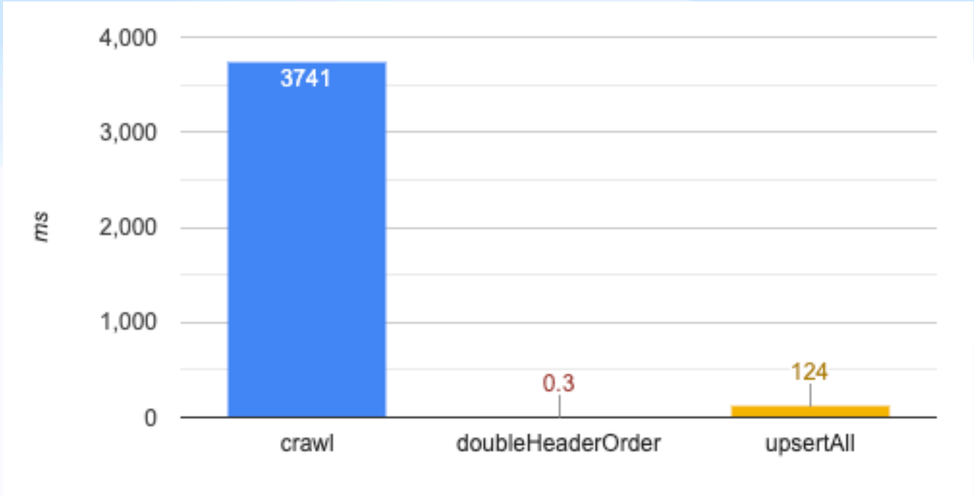
```
Hibernate:
    insert
    into
        games
        (away_pitcher, away_score, away_score_board_id, away_team_id, date, game_code, ga
         stadium_id, start_at)
    values
        (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
2025-10-05 00:40:13 [INFO] - [KboScoreboardService.fetchScoreboard] - [END TX] (3883ms)
2025-10-05 00:40:13 [INFO] - Request is completed (3930ms)
```

1. 성능 계측

쪼개서 확인해보기

```java
@Transactional
public ScoreboardResponse fetchScoreboard(final LocalDate date) {
    StopWatch sw = new StopWatch("scoreboard:" + date);
    sw.start("crawl");
    List<KboScoreboardGame> games = kboScoreboardCrawler.crawlScoreboard(date);
    sw.stop();

    sw.start("doubleHeaderOrder");
    applyDoubleHeaderOrder(games);
    sw.stop();

    sw.start("upsertAll");
    upsertAll(games, date);
    sw.stop();

    log.info("[SCOREBOARD] date={} phases={} total={}ms", date, sw.prettyPrint(), sw.getTotalTimeMillis());
    return new ScoreboardResponse(date, games);
}
```

```
2025-10-05 00:51:52 [INFO] - [SCOREBOARD] date=2025-10-04 phases=StopWatch 'scoreboard:2025-10-04': 4.001028526 seconds
---------------------------------------------
Seconds      %     Task name
---------------------------------------------
3.859164334  96%   crawl
0.000516292  00%   doubleHeaderOrder
0.141348     04%   upsertAll
 total=4001ms
2025-10-05 00:51:52 [INFO] - [KboScoreboardService.fetchScoreboard] - [END TX] (4003ms)
```

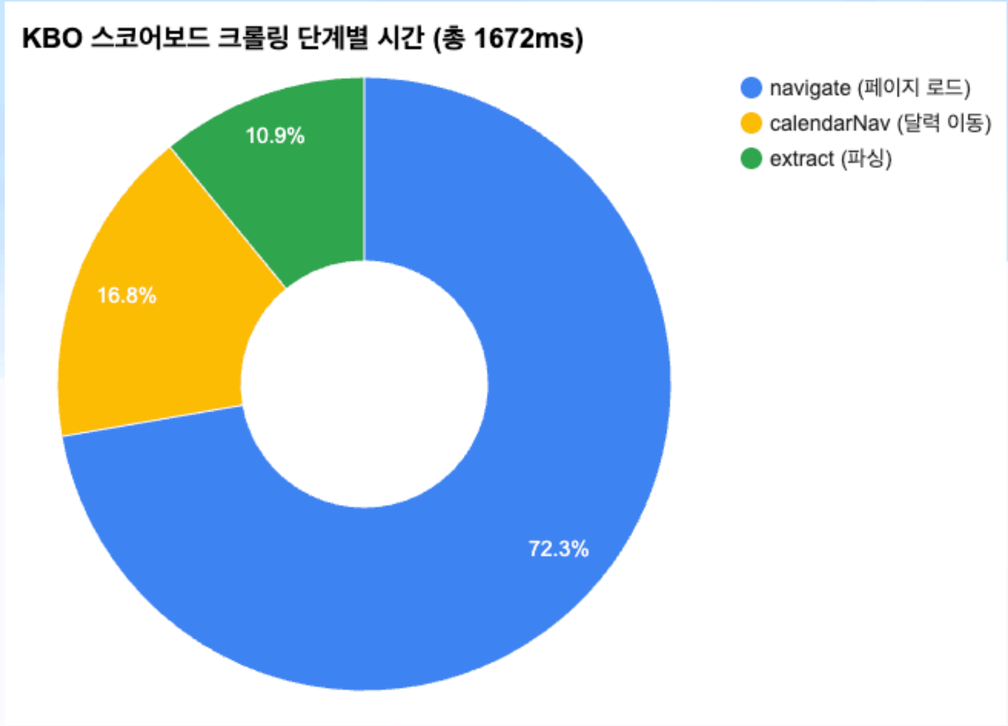


크롤링에서 96.8% 시간 소요

1. 성능 계측

크롤링 시간 단계별로 쪼개보기

```java
public List<KboScoreboardGame> crawlScoreboard(LocalDate date) {
    StopWatch sw = new StopWatch("crawl:" + date);
    try (Playwright pw = Playwright.create();
         Browser b = pw.chromium().launch(new BrowserType.LaunchOptions().setHeadless(true))) {

        sw.start("navigate");
        Page page = b.newPage();
        page.navigate(BASE_URL);              // 페이지 로드
        sw.stop();

        sw.start("calendarNav");
        navigateToDateUsingCalendar(page);    // 날짜 이동
        sw.stop();

        sw.start("extract");
        var games = extractScoreboards(page); // 파싱
        sw.stop();

        log.info("[CRAWL_OK] {} total={}ms\n{}", date, sw.getTotalTimeMillis(), sw.prettyPrint());
        return games;
    }
}
```



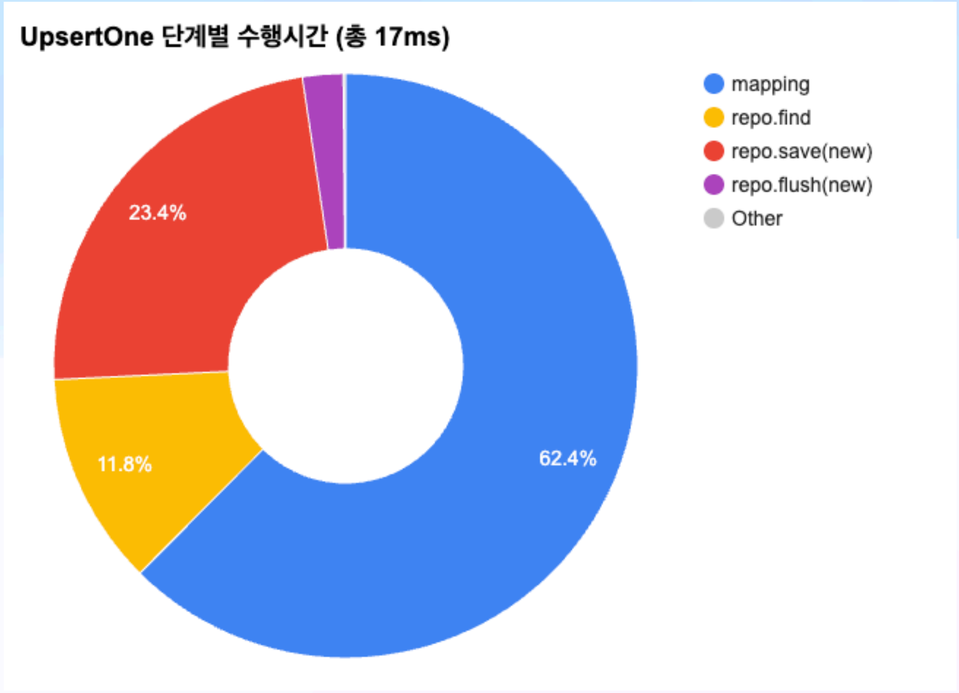
navigate (페이지 로드 / 렌더링)에서 병목

1. 성능 계측

DB 쓰기와 조회가 모두 발생하는 구간

Upsert 시간 단계별로 쪼개보기

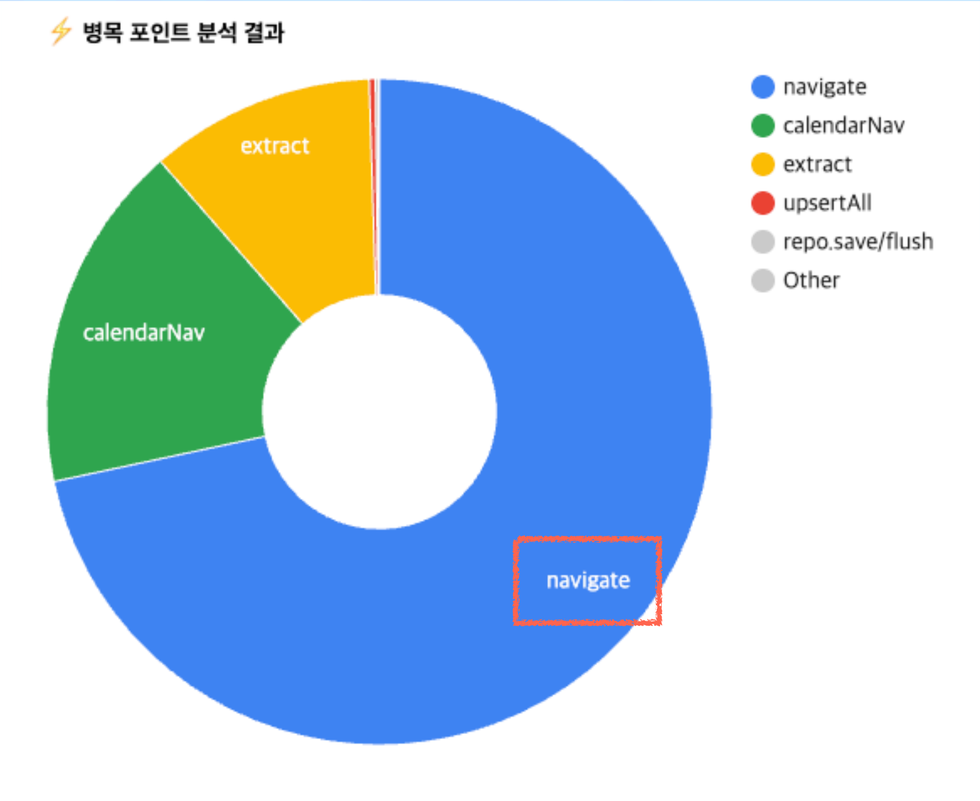
```java
private Game upsertOne(KboScoreboardGame dto, LocalDate date) {
    StopWatch sw = new StopWatch("upsertOne");

    sw.start("mapping");
    Team home = mapper.resolveTeam(dto.getHomeTeam());
    Team away = mapper.resolveTeam(dto.getAwayTeam());
    Stadium stadium = mapper.resolveStadium(dto.getStadium());
    sw.stop();

    sw.start("find or create");
    Game game = gameRepository.findByNaturalKey(date, home, away)
        .orElse(new Game(stadium, home, away, date));
    sw.stop();

    sw.start("update & save");
    game.updateFrom(dto);
    gameRepository.save(game);
    sw.stop();

    log.info("[UPSERT_ONE] total={}ms phases={}", sw.getTotalTimeMillis(), sw.prettyPrint());
    return game;
}
```



mapping(dto→entity 조회)에서 병목

1. 성능 계측

병목 포인트 분석 결과



🔷 가장 큰 원인 : 크롤링 - 페이지 로딩 대기

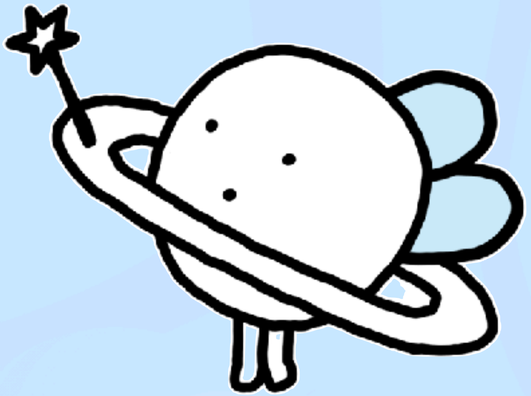
→ 렌더링 최적화하기

🔷 DB upsert는 병목 영향 ⬇

→ 저장보다 조회에서 많은 시간 소요

→ 쿼리 수 줄이고 안정성 개선하기

개요



1. 성능 계측
2. 크롤러 성능 개선
3. 운영 안정성 개선
4. 마무리

2. 크롤러 성능 개선

불필요한 리소스 요청 차단

불필요한 이미지, 영상 데이터를 다운로드하지 않도록 차단

```
page.route("**/*", route -> {
    String type = route.request().resourceType();
    // 필요 없는 리소스는 차단
    if ("image".equals(type) || "media".equals(type) || "font".equals(type)) {
        route.abort();
        return;
    }
    route.resume();
});
```

2. 크롤러 성능 개선

브라우저 생성 구조 개선

🔷 Before (문제 상황)

```
try (Playwright pw = Playwright.create();
     Browser browser = pw.chromium().launch(
         new BrowserType.LaunchOptions().setHeadless(true))) {

    Page page = browser.newPage();
    page.navigate(BASE_URL);

    // 크롤링
}
```

매 요청마다 브라우저 새로 생성

🔷 After

```
Page page = pwManager.acquirePage();
try {
    page.navigate(BASE_URL);
    navigateToDateUsingCalendar(page, date);

    // 크롤링
} finally {
    pwManager.releasePage(page);
}
```

브라우저 풀에서 Page만 빌려 사용

2. 크롤러 성능 개선

개선 결과

🔷 E2E 응답 시간 단축

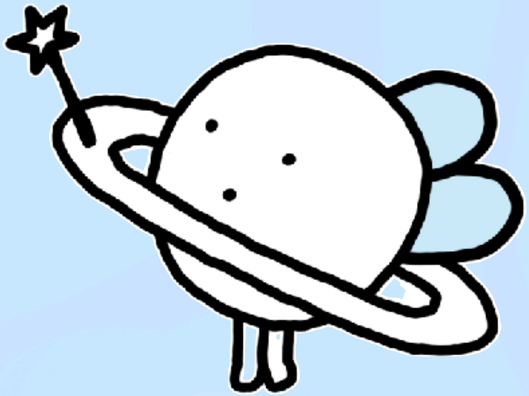
3865 ms → 1855 ms (-52.0%)

🔷 크롤링 - 렌더링 최적화

3741 ms → 1719 ms (-54.1%)

5분마다 반복되는 크롤링 작업에서는 시간 절감 효과 ⬆

개요

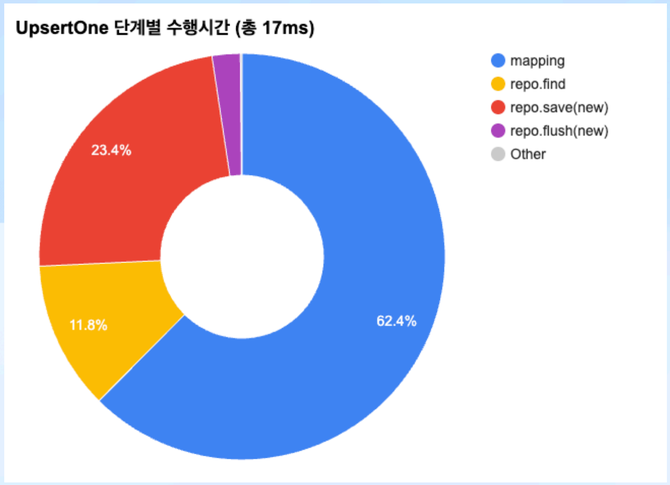


1. 성능 계측
2. 크롤러 성능 개선
3. 운영 안정성 개선
4. 마무리

3. 운영 안정성 개선

날짜별 반복 쿼리

단건마다 UPSERT 반복하여 쿼리 폭증 / DB 오버헤드



mapping(조회) >> save

```
2025-10-05 10:05:14 [INFO] - [UPSERT_ONE] key=2024-04-07|NC vs SSG status=CREATE
-------------------------------------------------
Seconds     %     Task name
-------------------------------------------------
0.006087834  40%  mapping
0.0015165    10%  repo.find
0.000029958  00%  create.entity
0.007317525  48%  repo.save(new)
0.000392167  03%  repo.flush(new)

Hibernate:
    select
        t1_0.team_id,
        t1_0.name,
        t1_0.short_name,
        t1_0.team_code
    from
        teams t1_0
    where
        t1_0.short_name=?
Hibernate:
    select
        t1_0.team_id,
        t1_0.name,
        t1_0.short_name,
        t1_0.team_code
    from
        teams t1_0
    where
        t1_0.short_name=?
```

```
2025-10-05 11:03:12 [INFO] - [UPSERT_ONE] key=2024-04-07|KIA vs 삼성 status=CREATE
-------------------------------------------------
Seconds     %     Task name
-------------------------------------------------
0.005837542  29%  mapping
0.006861208  34%  repo.find
0.000031709  00%  create.entity
0.007090041  35%  repo.save(new)
0.000428458  02%  repo.flush(new)

Hibernate:
    select
        t1_0.team_id,
        t1_0.name,
        t1_0.short_name,
        t1_0.team_code
    from
        teams t1_0
    where
        t1_0.short_name=?
Hibernate:
    select
        t1_0.team_id,
        t1_0.name,
        t1_0.short_name,
        t1_0.team_code
    from
        teams t1_0
    where
        t1_0.short_name=?
```

3. 운영 안정성 개선

팀 · 구장 캐싱

Before : 같은 팀, 구장 매번 SELECT

```
private Game upsertOne(KboScoreboardGame dto, LocalDate date) {
    // 매핑 (매 행마다 SELECT 발생)
    Team away = teamRepository.findByShortName(dto.getAwayTeam().name())
            .orElseThrow(() -> new NotFoundException("Unknown team: " + dto.getAwayTeam().name()));
    Team home = teamRepository.findByShortName(dto.getHomeTeam().name())
            .orElseThrow(() -> new IllegalArgumentException("Unknown team: " + dto.getHomeTeam().name()));
    Stadium stadium = stadiumRepository.findByLocation(dto.getStadium())
            .orElseThrow(() -> new IllegalArgumentException("Unknown stadium: " + dto.getStadium()));
    // upsert
}
```

After : 한번만 로딩하여 Map 조회

```
// 1) 팀/구장 1회 로딩
Map<String, Team> teamByShort = teamRepository.findAll().stream()
        .collect(toMap(Team::getShortName, Function.identity()));
Map<String, Stadium> stadiumByLocation = stadiumRepository.findAll().stream()
        .collect(toMap(Stadium::getLocation, Function.identity()));

// 2) 행 변환 (메모리 조회만 수행)
List<GameJdbcBatchUpsertRepository.GameUpsertRow> rows = new ArrayList<>(list.size());
for (KboScoreboardGame dto : list) {
    Team away = teamByShort.get(dto.getAwayTeam().name());
    Team home = teamByShort.get(dto.getHomeTeam().name());
    Stadium stadium = stadiumByLocation.get(dto.getStadium());
    // upsert
}
```

After : 결과

```
2025-10-05 11:07:21 [DEBUG] - 조회 날짜: 2024-04-07
2025-10-05 11:07:22 [DEBUG] - 조회 날짜: 2024-04-08
2025-10-05 11:07:25 [DEBUG] - 조회 날짜: 2024-04-09
2025-10-05 11:07:26 [DEBUG] - 조회 날짜: 2024-04-10
Hibernate:
    select
        t1_0.team_id,
        t1_0.name,
        t1_0.short_name,
        t1_0.team_code
    from
        teams t1_0
Hibernate:
    select
        s1_0.stadium_id,
        s1_0.full_name,
        s1_0.latitude,
        s1_0.location,
        s1_0.longitude,
        s1_0.short_name
    from
        stadiums s1_0
2025-10-05 11:07:27 [INFO] - [UPSERT-BATCH/ALL] period=2024-04-07~2024-04-10 rows=15 success=15 failed=0 took=7ms
```

최초 1회만 수행

3. 운영 안정성 개선

배치 처리?

🔷 실시간 수집

하루에 5경기 정도 5분마다 조회

→ 성능 이점 적음

🔷 대량 데이터 적재

시즌별, 연도별 경기 데이터를 한번에 적재 시 사용

→ 스키마 변경, 정합성

→ 수천건의 데이터 → 성능 이점 큼

3. 운영 안정성 개선

Jdbc Batch Update

여러 개의 SQL 문을 한 번의 네트워크 전송으로 DB에 보내는 기능

🔷 Before

```
for (...) {
  jdbcTemplate.update("INSERT ...", ...);
}
```

매번 DB에 요청 전송 → 네트워크 왕복 ⬆

🔷 After

```
PreparedStatement ps = connection.prepareStatement(SQL);
for (Record r : records) {
    // 파라미터 세팅만 반복
    ps.setString(1, r.code());
    ps.addBatch();
}
ps.executeBatch(); // 한 번에 DB로 전송
```

여러 SQL을 한번에 DB로 전송하여 DB 통신 횟수를 줄임

3. 운영 안정성 개선

Jdbc Batch Update 도입

🔷 rewriteBatchedStatements 설정

```yaml
spring:
  datasource:
    url: jdbc:mysql://<host>:<port>/<db>?rewriteBatchedStatements=true
```

addBatch() → 다중행 SQL로 재작성

🔷 Batch Size (chunk) 결정

```
private static final int BATCH_SIZE = 300;

private static final String UPSERT_SQL = """
    INSERT INTO games (game_code, home_team_id, away_team_id, date, start_at,
                       home_score, away_score, home_pitcher, away_pitcher, game_state)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    ON DUPLICATE KEY UPDATE
        home_score = VALUES(home_score),
        away_score = VALUES(away_score),
        game_state = VALUES(game_state)
    """;
```

1년 야구경기 720개 → chunk : 300개

🔷 파라미터 값 대입

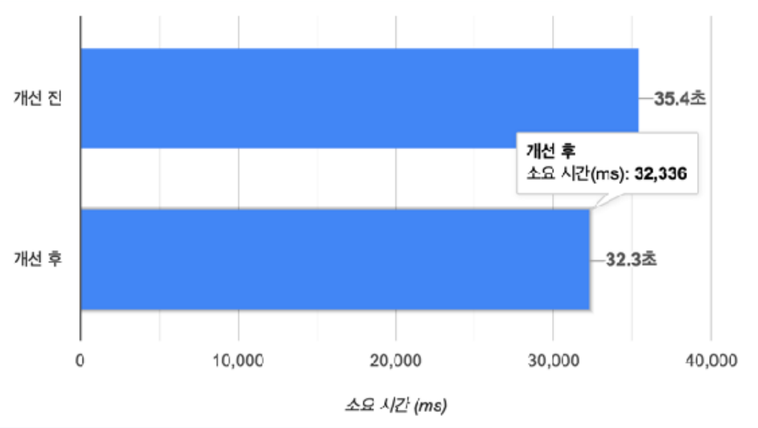
```
@Transactional
public void batchUpsert(List<GameUpsertRow> rows) {
    for (int from = 0; from < rows.size(); from += BATCH_SIZE) {
        int to = Math.min(from + BATCH_SIZE, rows.size());
        List<GameUpsertRow> chunk = rows.subList(from, to);

        jdbcTemplate.batchUpdate(UPSERT_SQL, new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i) throws SQLException {
                GameUpsertRow r = chunk.get(i);
                ps.setString(1, r.gameCode());
                ps.setLong(2, r.homeTeamId());
                ps.setLong(3, r.awayTeamId());
                ps.setObject(4, r.date());
                ps.setObject(5, r.startAt());
                ps.setInt(6, r.homeScore());
                ps.setInt(7, r.awayScore());
                ps.setString(8, r.homePitcher());
                ps.setString(9, r.awayPitcher());
                ps.setString(10, r.gameState());
            }

            @Override
            public int getBatchSize() {
                return chunk.size();
            }
        });
    }
}
```

3. 운영 안정성 개선

개선 결과

속도 뿐 아니라 안정적으로 대량 데이터 처리 가능

🔷 E2E 응답 시간 (한달)



35.4 s → 32.3 s (-8.7%)

🔷 DB 쿼리 횟수

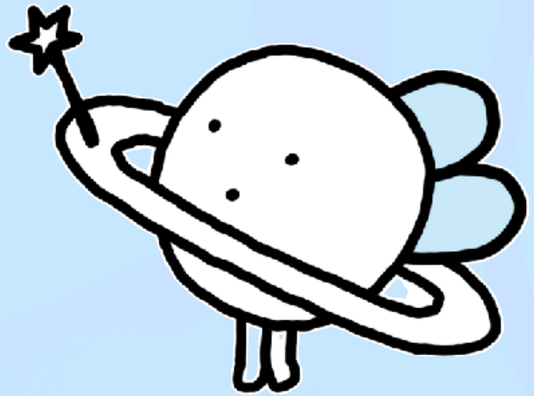
N회 → 1회 (1/300)

🔷 실패 처리

전부 롤백

→ 청크마다 커밋하므로 재시도 가능

개요



1. 성능 계측
2. 크롤러 성능 개선
3. 운영 안정성 개선
4. 마무리

마무리

작업 목표 : KBO 크롤러의 성능과 안정성 높이기

🔷 크롤링 - 렌더링 최적화

불필요한 리소스 차단, 브라우저 재사용

→ 50% 이상 시간 단축 (3.9s → 1.9s)

🔷 저장 단계 (Upsert)

메모리 캐싱 + Jdbc batch update

→ 대량 데이터 적재 안정성 향상

꼿