# MZ 세대 위한 1인 가구 추천 서비스

**Redis 기반 실시간 추천 엔진 + Oracle 성능 최적화**

https://github.com/bumjinDev/wherehouse_SpringBoot?tab=readme-ov-file

# 기획 의도 및 기술 도전 과제

**비즈니스 문제**

## MZ 1인 가구 750만 시대, 주거 선택 시 정보 부재

범죄율, 편의시설, 가격을 통합 분석하는 맞춤형 통합 서비스 필요.

## 기술 도전 과제

### Challenge 1

#### 실시간 다중 조건 필터

**25개 자치구 × 수만 건 매물을 가격·평수·안전성 3개 조건으로 동시 필터링**

→ 매 요청마다 Oracle 조회 시 응답 지연 + Connection Pool 경합으로 동시 사용자 증가 시 서비스 응답 불가

**해결 방향**

Redis Sorted Set에 조건별 인덱스를 사전 구축하여 DB 접근 없이 메모리 내 교집합 연산으로 필터링 완료

— 매물 부족 시 2단계 Fallback으로 조건을 순차 완화하여 "검색 실패" 방지

목표 : 추천 요청 경로에서 Oracle 조회 0회, 메모리 내 필터링 완료

### Challenge 2

#### 외부 API 병목

**Kakao Map API 15개 카테고리를 WebClient.block() 동기 방식으로 순차 실행**

→ 외부 API 구간이 전체 응답의 92.2% 차지 (API 1,221ms / 전체 1,324ms)
지도 클릭 후 편의시설 정보 로딩에 1.3초 이상 소요

**해결 방향**

CompletableFuture + I/O 전용 ThreadPoolExecutor으로 15개 호출을 병렬 실행

목표 : 외부 API 호출 시간 79% 감소, 총 응답 시간 72% 감소

### Challenge 3

#### Oracle 실행 계획 분석

**N+1 패턴으로 요청당 최대 25회 DB 호출**

→ Connection Pool(6개) 대비 동시 요청 시 경합 발생

→ 사용자 응답 지연 직결

**해결 방향**

IN절 Chunk 크기 고정으로 SQL 템플릿 통일

— Bulk Fetch / Chunk 1000 / 22 / 61 4가지 대안을 V$SQL 기반 정량 분석으로 실험 후 Index Scan 유지 + 경합 0%인 Chunk 61 선택

목표 : Soft Parse 88.5% + Index Scan + Connection 경합 0%

**해결 방향**

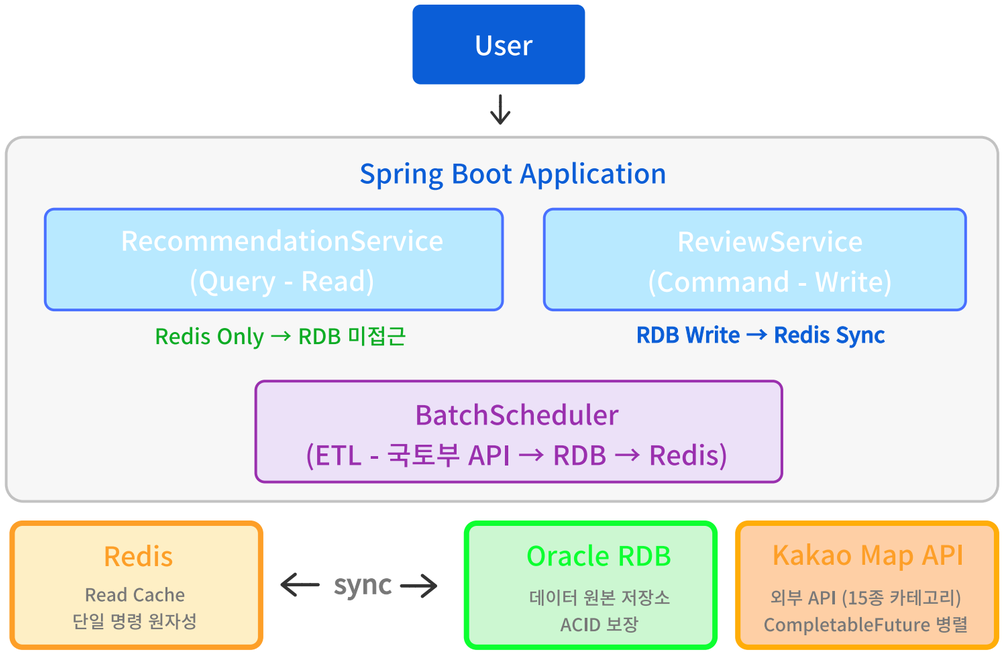
추천 응답 경로 : Oracle 조회 0회 (Redis 메모리 내 완료)

외부 API 응답 : 1,324ms → 367ms (72.3% 감소)

DB 쿼리 최적화 : Connection 경합 47.9% → 0%, Soft Parse 88.5%

# 시스템 아키텍처

CQRS + Hybrid Storage 전략




## 핵심 서비스 로직

### Redis 실시간 검색 엔진

Sorted Set 교집합 연산으로 가격/평수/안전성 다중 조건 필터링

→ DB 조회 없이 메모리 내 완료

### 2단계 Fallback 로직

엄격 검색(전 조건 충족) → 확장 검색(3순위→2순위→1순위 조건 순차 완화)

→ "검색 실패" 대신 "현실적 대안 제시"로 서비스 이탈 방지

### 하이브리드 추천

정량 점수(Redis 교집합) + 정성 점수(RDB 리뷰 통계) 가중 합산

→ reviewCount ≥ 5: 50:50 / reviewCount < 5: 정량 100%

## 설계 결정 + Why

### 1 CQRS 패턴 (Read/Write 물리 분리)

**Why :** 추천 조회(Read)는 초당 다수 요청을 저지연으로 처리해야 하고, 리뷰 작성(Write)은 ACID 트랜잭션 보장이 필요하다. 두 경로의 성능 특성과 일관성 요구사항이 다르므로, 읽기 경로는 Redis 메모리 연산으로, 쓰기 경로는 Oracle RDB 트랜잭션으로 물리적 분리하였다.

**효과 :** 추천 요청 경로에서 Oracle 조회 하지 않음.

### 2 Write-Through 캐시 동기화

**Why :** CQRS로 Read/Write 저장소를 분리하면, Redis(Read)와 Oracle(Write)의 데이터 정합성 문제가 발생한다. Write-Through 방식은 RDB 트랜잭션 커밋 직후 Redis를 동기 갱신하므로, 최종 일관성이 아닌 즉시 일관성을 보장한다.

**효과 :** 리뷰 작성 후 추천 결과에 즉시 반영 — 사용자 체감 정합성 보장

### 3 외부 API 격리 (CompletableFuture + ThreadPoolExecutor)

**Why :** Kakao Map API 호출은 네트워크 I/O 의존으로 지연 시간이 불확실하다. I/O 전용 ThreadPoolExecutor(20스레드)로 격리하여 Main 스레드의 블록킹을 뮤재룰 해결함으로써 외부 API 장애가 추천/리뷰 서비스에 전파되지 않도록 하였다.

**효과 :** 외부 API 응답 시간 72.3% 감소 (1,324ms → 367ms)

## 핵심 서비스 로직

1. 추천 조회

   User → RecommendationService → Redis (RDB 무접근)

2. 리뷰 작성

   User → ReviewService → RDB → Redis 동기화

3. 배치 갱신

   BatchScheduler → 국토부 API → RDB → Redis

4. 위치 분석

   User → Kakao API (CompletableFuture 병렬)

# 트러블슈팅 #1: 외부 API 병목 해결

CompletableFuture 기반 비동기 병렬 처리로 72.3% 응답 시간 단축

## 문제 정의

**Kakao Map API 15개 카테고리 순차 호출 — 전체 응답의 92.2%가 외부 API 구간**

**WebClient.block() 동기 방식으로 15개 카테고리를 순차 실행. 각 호출 완료까지 Main 스레드가 블로킹되어 카테고리** 수에 비례하여 응답 시간이 선형 증가.

외부 API : 1,221ms / 전체 : 1,324ms → 비중 : 92.2% 사용자 체감 : **편의시설 정보 로딩 1.3초 이상 소요**

## 대안 검토

| 대안 | 장점 | 단점 / 기각 사유 |
|---|---|---|
| WebClient Reactive | 논블로킹 I/O 스레드 효율 최고 | 기존 동기 코드베이스 전면 리팩토링 필요. Service/Repository 반환 타입 전환 불가피 → 프로젝트 일정 대비 과도한 변경 범위 |
| @Async + Future | Spring 기본 제공 설정 단순 | Future.get() 반환값 합산 불편 하며 다수 Future 완료 대기 조합 API 부재. 예외 시 ExecutionException 래핑 복잡 |
| **CompletableFuture + ThreadPoolExecutor** | 기존 동기 코드에 supplyAsync() 래핑으로 적용. allOf() 대기, exceptionally() 장애 격리 | 스레드 풀 크기 설계 필요 (I/O 바운드 특성 고려) |

## 트레이드오프 + 선택 근거

### WebClient Reactive 기각

현재 동기 기반(Spring MVC) 코드베이스에서 Reactive 전환은 전면적으로 재설계해야 한다. 외부 API 1개 메서드 개선을 위해 프로젝트 전체 구조를 변경하는 것은 변경 범위 대비 효과가 과도하다.

### CompletableFuture 선택

기존 동기 메서드를 supplyAsync()로 감싸는 것만으로 비동기 전환 가능. 15개 호출의 완료 대기는 allOf()로 단일 지점에서 합산하며, 개별 실패는 exceptionally()로 빈 리스트를 반환하여 전체 응답 실패를 방지한다.

## 정량적 성과

| 요청 구분 | 개선 전 | 개선 후 | 개선율 |
|---|---|---|---|
| 외부 API 호출 평균 | 1,221.0ms | 254.9ms | -79.1% |
| 외부 API 호출 p95 | 1,388.8ms | 782.9ms | -43.6% |
| 총 응답 시간 | 1,324.2ms | 366.9ms | -72.3% |

사용자 체감 개선 : **편의시설 정보 로딩 1.3초 → 0.4초**

## 처리 흐름 비교

**Before** (순차 실행) Cat1 → 대기 → Cat2 → 대기 → ... → Cat15 → 합산

15개 RTT 순차 누적 = 1,221ms

▼

After (병렬 실행) supplyAsync(EXECUTOR) × 15 → 병렬 → allOf() 대기 → 합산

가장 느린 1개 카테고리 응답 시간 ≈ 255ms

## 코드 핵심부

```
// 15개 카테고리 병렬 검색 — CompletableFuture + I/O 전용 ThreadPoolExecutor(20)
Map<String, CompletableFuture<List<Map<String, Object>>>> futureMap = new HashMap<>();

for (String category : categories) {
    futureMap.put(category, CompletableFuture.supplyAsync(() ->
            searchPlacesByCategory(latitude, longitude, category, radius),
            KAKAO_API_EXECUTOR                          // I/O 전용 스레드 풀
    ).exceptionally(ex -> new ArrayList<>()));          // 장애 격리: 실패 시 빈 리스트
}

CompletableFuture.allOf(futureMap.values().toArray(new CompletableFuture[0])).join();  // 전체 완료 대기
```

p95(783ms)가 평균(255ms) 대비 높은 이유: 특정 카테고리의 Kakao API 응답 지연이 병렬 그룹 전체의 완료 시간을 결정한다.

# 트러블슈팅 #3: Oracle IN절 Chunking 최적화

## N+1 → Bulk Fetch → Chunk 61, V$SQL 기반 3단계 실행계획 분석

### 문제 정의 : N+1 패턴

25개 자치구 순회 시 자치구당 findAllById() 호출 — 최대 25회 DB 호출 (N+1 패턴)

V$SQL 분석: 78개 고유 SQL_ID(Hard Parse), 340회 실행, Cost 편차 6594(99배)

원인 : 자치구별 매물 수 541,000개 가변 → IN절 바인드 변수 개수 매번 상이 → Oracle이 별개 SQL로 인식

HikariCP Pool 6 대비 동시 20요청 : Waiting 최대 6건, NOT_ADDED 162회 → Connection 경합

### 1차 해결 및 한계

**Bulk Fetch 적용**

전체 ID를 단일 IN절로 병합 → N+1(최대 25회) → 1회 호출

RDB 호출 25회→1회(−96%), 전체 응답 293ms→191ms(−34.9%)

**V$SQL 분석으로 확인된 한계**

20개 요청 → 20개 고유 SQL_ID. 모든 SQL: EXECUTIONS=1, Soft Parsing_CALLS=1

→ Library Cache 재사용 전무 (100% Hard Parse)

요청별 매물 수 가변 → IN절 바인드 변수 개수 상이 → Soft Parse 불가

### 트레이드오프 + 선택 근거

2차 해결: IN절 바인드 변수 개수를 고정 → 동일 SQL_ID 재사용 유도. 3가지 Chunk(1000/22/61)를 V$SQL, V$PLAN, Connection Pool 로 실측 비교.

#### Chk 1000 — 기각

TABLE ACCESS FULL 동작 및 Predicate: filter(후처리) — 인덱스 경로 제외

SQL 호출 21회 최소이나, Index 무시 → 데이터 증가 시 성능 열화 예측 불가

#### Chk 22 — 기각

INLIST ITERATOR + INDEX UNIQUE SCAN 동작

Predicate: access(접근 경로). Soft Parse 82.8%, Cost 27 — DBMS 효율 최고

SQL 호출 381회 급증 → waiting 최대 10, 심각 경합(≥7) 47.9%

총 Cost 27×381=10,287 과대. Pool 6 환경에서 Connection 고갈 위험

#### Chk 61 — 선택

INLIST ITERATOR + INDEX UNIQUE SCAN 유지

SQL 호출 134회(Chk22 대비 −64.8%), 심각 경합 0% 완전 해소

총 Cost 68×134=9,112 — Chk22(10,287) 대비 11% 낮음

RDB 시간 비율 48.5%→17.3% 회복. Soft Parse 88.5%

응답 318ms는 DBMS가 아닌 Redis 순차호출 기인 하므로 DBMS 병목이 아닌 별도의 트러블 슈팅 필요

### Chunk 크기별 실측 비교 (표)

| 지표 | Bulk | Chk 1000 | Chk 22 | Chk 61 |
|---|---|---|---|---|
| 실행계획 | - | Full Scan | Index Scan | Index Scan |
| SQL 호출 | 20회 | 21회 | **381회** | 134회 |
| SQL_ID 수 (Hard Parse) | 20 | 20 | 11 | 15 |
| Soft Parse 비율 | 0% | 4.8% | 96.8% | 88.5% |
| 경합(max waiting) | 최대 2 | 최대 1 | **최대 10** | 최대 4 |
| 심각 경합(≥7) | 0% | 0% | **47.9%** | 0 |
| 총 Cost(Cost×호출) | — | 4,011 | 10,287 | 9,112 |
| 응답 시간(평균) | 191ms | 178ms | 268ms | 318ms |

** Bulk는 요청별 바인드 변수 개수가 54~1,276개로 가변이므로 SQL_ID마다 Cost가 상이하여 대표값 표기 불가.
Chk 계열은 표준 Chunk SQL의 바인드 변수 개수가 고정되어 Cost가 단일 값으로 수렴.

### 실행 계획 비교

**Chunk 1000**

```
SELECT STATEMENT
└── TABLE ACCESS FULL
    (REVIEW_STATISTICS)
```

Predicate: filter (후처리)

Cost: 191

**Chunk 22 / 61**

```
SELECT STATEMENT
└── INLIST ITERATOR
    └── TABLE ACCESS BY INDEX ROWID
        └── INDEX UNIQUE SCAN
            (PK_REVIEW_STATISTICS)
```

Predicate: access (접근 경로)

Cost: 27(Chk22) / 68(Chk61)

### 결론 : Chunk 61 선택

확장성 : Index Scan 유지 — 데이터 증가에도 실행계획 안정

안정성 심각 경합(waiting≥7) 47.9% → 0% 완전 해소

캐시 효율 Soft Parse 88.5%, Hard Parse +42 (Chk22의 1/4)

총 비용 Cost×호출 = 9,112 (Chk22 10,287 대비 −11%)

# 트러블슈팅 #6: Redis Pipeline RTT 최적화

순차 Redis 호출 → Pipeline 일괄 전송으로 RTT 66.7% 절감, 지역구당 응답 시간 54.4% 단축

## 문제 정의

주거지 추천 요청 시, 서울 25개 자치구별로 보증금·월세·평수 3개 조건의 Redis Sorted Set 범위 검색 (ZRANGEBYSCORE)을 순차 동기 호출하는 구조

**지역구당 3회 RTT × 25 자치구 = 75회 네트워크 왕복**

→ 각 Command 응답 수신 완료까지 스레드 블로킹, RTT가 선형으로 누적되는 구조

I/O 비율 99.7% (Layer 1 실측) — 전형적 I/O 바운드 워크로드
루프 총 시간 6,540.6ms | 지역구 평균 응답 128.95ms (강남구 제외)

## Pipeline 적용 근거

| 99.7% | 31 : 36 : 33 | READ |
|---|---|---|
| I/O 비율 | Command Latency 비율 | 읽기 전용 워크로드 |
| MethodTime의 99.7%가 Redis 응답 대기 시간 → I/O 바운드 확정 | 3개 Command가 총 Latency에 균등 기여 특정 Command 편중 없이 묶기에 적합 | 3개 ZRANGEBYSCORE 모두 읽기 전용 Pipeline의 원자성 미보장이 문제되지 않음 |

→ 3개 조건 모두 충족: Pipeline은 이 워크로드의 RTT 누적 구조를 제거하는 데 정확히 적합한 기법

## 선택 근거 + 동작 원리

executePipelined()의 RedisCallback 내부에서 3개 connection.zRangeByScore()를 write buffer에 순차 적재한 뒤, 콜백 반환 시점에 단일 TCP Write로 일괄 flush한다.

Redis 서버는 수신한 3개 Command를 순서대로 처리하고, 3개 RESP Array 응답을 하나의 TCP 스트림으로 반환한다. 클라이언트는 1회 네트워크 왕복으로 3개의 응답을 수신.

추가 스레드 풀 없이 단일 TCP 연결에서 동작하며, 기존 opsForZSet() API 구조를 low-level RedisCallback으로 전환하는 최소 변경만으로 적용 가능하다.

**결론 : Pipeline 선택 — RTT 횟수만 1/3로 절감, 1회 왕복 비용 증가 없이 구조적 개선**

## 처리 흐름 비교

**Before (순차 실행)** Cmd1 → 대기 → Cmd2 → 대기 → Cmd3 → 대기 → 합산
**3회 RTT × 25 자치구 = 75회 왕복 → 순차 누적 = 6,540.6ms**

▼

After (병렬 전송) Cmd 3개 일괄 전송 (1 TCP Write) → 응답 일괄 수신 → 합산
1회 RTT × 25 자치구 = 25회 왕복 → 4,320.0ms (34.0% 감소)

## 검증 : Wireshark TCP 패킷 분석

| 측정 지표 | 순차 호출 (Before) | Pipeline (After) |
|---|---|---|
| 단일 Req-Resp Latency | 38.5 ms (평균) | 39.3 ms (평균) |
| 요청 패킷 수 | 75개 | 25개 |
| 응답 패킷 수 / 바이트 | 3,852개 / 5.0 MB | 3,856개 / 5.0 MB |

→ Pipeline 1회 왕복 비용(39.3ms) ≈ 순차 단일 명령 RTT(38.5ms). RTT 횟수만 1/3로 절감, 1회 왕복 비용 증가 없음을 패킷 레벨에서 입증

## 정량적 성과 (Before → After)

| 지표 | 순차 호출 (Before) | Pipeline (After) | 개선율 |
|---|---|---|---|
| 25개 자치구 전체 검색 시간 총합 | 6,540.6 ms | 4,320.0 ms | -34.0% |
| 지역구별 평균 응답 (강남구 제외) | 128.95 ms | 58.80 ms | -54.4% |
| 네트워크 왕복 횟수 (RTT) | 75회 | 25회 | -66.7% |
| 지역구당 네트워크 대기 (Wireshark 실측) | 38.5ms × 3 = 115.5 ms | 39.3ms × 1 = 39.3 ms | -66.0% |

## 강남구 이상치 분석

103,470건(4.0MB) 응답 payload → MethodTime 3,429ms → 2,895ms (15.6% 감소)
RTT 절감분(65.8ms)은 전체 대비 2.2%에 불과 → 병목 본질은 payload 크기이며, 별도 알고리즘 개선 필요

**핵심 인사이트** Pipeline은 RTT 횟수를 1/3로 줄이면서 1회 왕복 비용을 증가시키지 않는 구조적 개선이다. Layer 1(nanoTime) + Layer 2(Wireshark) 2계층 실측으로 입증.

# 트러블슈팅 #4: LIKE 검색 Index 최적화

Full Table Scan → Index Range Scan 전환

## 문제 정의

**findPropertyIdsByName** 쿼리가 전체 응답 시간의 **66.3%~86.5%** 차지
LIKE '%keyword%' 패턴으로 Full Table Scan 발생

→ 쿼리 평균 26.3ms / 메서드 45.7ms
→ 50 동시 요청 슬로우 쿼리(≥100ms) **103건**

## 원인 분석

선행 와일드카드 '%keyword%'는 B-Tree 인덱스의 정렬 구조를 활용할 수 없다. B-Tree는 키 값을 좌측부터 정렬하므로, 검색 시작점(leftmost prefix)을 특정할 수 없으면 Range Scan 시작 위치를 결정 불가.

**실행계획** : CHARTER/MONTHLY 각 TABLE ACCESS FULL, Cost 68+68=137
**→ Filter Predicates: 데이터 전량 읽은 후 조건 필터링 (비효율)**

## 실행계획 비교

개선 전

SELECT STATEMENT
└─ **TABLE ACCESS FULL**
**Predicate: filter (후처리)**
**Cost: 137**

개선 후

SELECT STATEMENT
└─ **INDEX RANGE SCAN**
**Predicate: access (접근 경로)**
**Cost: 5**

### 결론: LIKE 'keyword%' + B-Tree Index 전환

**확장성 :** 데이터 증가에도 Index Range Scan 유지 — **Cost 96% 감소 (137→5)**
**안정성 :** 슬로우 쿼리(≥100ms) 103건 → **0건 완전 제거**
**트레이드오프 :** 후방 검색 불가 — 실제 사용 패턴에서 영향 미미

## 대안 검토

| 대안 | 장점 | 단점 / 기각 사유 |
|---|---|---|
| Oracle Text (Full-Text Index) | 양방향 LIKE 대체 가능 중간 문자열 검색 지원 | 별도 역인덱스 구조 생성·동기화 관리 필요 및 1인 프로젝트 운영 복잡도 대비 과도 |
| LIKE 'keyword%' + apt_nm 인덱스 ✔ **선택** | 기존 B-Tree 인덱스 적용 확인 | 후방 검색 불가 (기능적 제약) |

## 트레이드오프 + 선택 근거

아파트명 검색의 실제 사용 패턴: 사용자는 "래미안", "푸르지오" 등 앞부분을 입력하는 경우가 대다수. 중간 문자열 검색은 극소수. 전방 일치 전환의 기능적 제약은 실제 사용자 경험에 영향 미미.

| 검색어 '삼성' 입력 시 | 부분 일치 (개선 전) | 전방 일치 (개선 후) |
|---|---|---|
| 래미안**삼성** | ✓ | ✗ |
| **삼성**래미안 | ✓ | ✓ |

## 정량적 성과

| 항목 | 개선 전 | 개선 후 | 개선율 |
|---|---|---|---|
| 쿼리 소요시간(평균) | 26.3ms | 6.0ms | 77.2% |
| 전체 메서드 소요시간 | 45.7ms | 22.4ms | 51.0% ↓ |
| 병목 비중 | 66.3% | 22.0% | 66.8% ↓ |
| 슬로우 쿼리 (≥100ms) | 103건 | 0건 | 100% ↓ |

# 트러블슈팅 #5: OOM 병목 Slice 기반 청크 처리

findAll() 일괄 로드 → Slice 페이징으로 힙 피크 72.7% 감소

## 문제 정의

BatchScheduler 11만 건 findAll() 일괄 로드
→ JPA가 전체 ResultSet을 단일 List에 적재 + Entity + 영속성 컨텍스트 메타데이터 GC 불가 상태 유지
→ 힙 피크 점유율 64.4%, 엔티티 메모리 합계 99.2MB (Charter 50.3 + Monthly 48.9)
→ 데이터 비례 선형 증가 구조 — 350,000건 이상 시 OOM 발생

## 대안 검토

| 대안 | 장점 | 단점 / 기각 사유 |
|---|---|---|
| Page<T> 기반 / 청크 처리 | 전체 건수<br>페이지 수 제공<br>getTotalElements, getTotalPages) | 청크마다 COUNT(*) 쿼리 자동<br>실행으로 인해 불필요한 오버헤드 발생 |
| Slice<T> 기반 청크 처리 ✔ **선택** | COUNT 쿼리 없이 LIMIT N+1로<br>다음 페이지 유무 판단 | OFFSET 페이징 오버헤드 |

## 트레이드오프 + 선택 근거

Page<T>는 매 청크 요청마다 COUNT(*) 쿼리를 자동 실행한다. 배치 동기화에서 전체 건수 정보는 비즈니스적으로 불필요하므로, 이 COUNT 쿼리는 순수 오버헤드가 된다. 현재 6청크 규모에서도 6회의 COUNT 쿼리가 추가 발생하며, 데이터 증가 시 비용이 비례하여 증가한다.

Slice<T>는 LIMIT N+1 기법으로 다음 청크 존재 여부를 판단하므로 COUNT 쿼리가 완전히 제거된다. DB 로드 시간은 증가(1,727ms → 3,553ms)하지만, 힙 피크가 청크 크기에만 비례하므로 데이터 증가에도 안정적이다.

배치 작업 특성상 실행 시간 증가(7초 → 12초)는 허용 범위이며, **OOM으로 인한 배치 전체 실패** vs 처리 시간 63% 증가는 명확한 트레이드오프이다. 메모리 안정성을 우선하여 Slice 방식을 선택하였다.

### 결론 : Slice 청크 처리 선택

**확장성 :** 데이터 증가해도 힙 피크 일정 유지 (청크 크기에만 비례)

**안정성 :** 힙 점유율 64.4% → 17.6%, 심각 경합 완전 해소

**허용 범위 :** 배치 7초 → 12초 증가는 안정성을 위한 트레이드 오프

### 핵심 인사이트

OOM 발생 시 전체 배치 실패 vs 처리 시간 63% 증가는 명확한 트레이드오프 발생하나 배치 작업 특성상 시간 증가보다 메모리 안전성 확보가 우선.

## 정량적 성과

| 지표 | 개선 전 | 개선 후 | 개선율 |
|---|---|---|---|
| 엔티티 메모리 합계 | 99.2 MB | 32.5 MB | -67.2% |
| 힙 피크 점유율 | 64.4% | 17.6% | -72.7% |
| 전세 엔티티 점유 메모리 | 50.3 MB | 17.2 MB | -65.8% |
| 월세 엔티티 점유 메모리 | 48.9 MB | 15.3 MB | -68.7% |

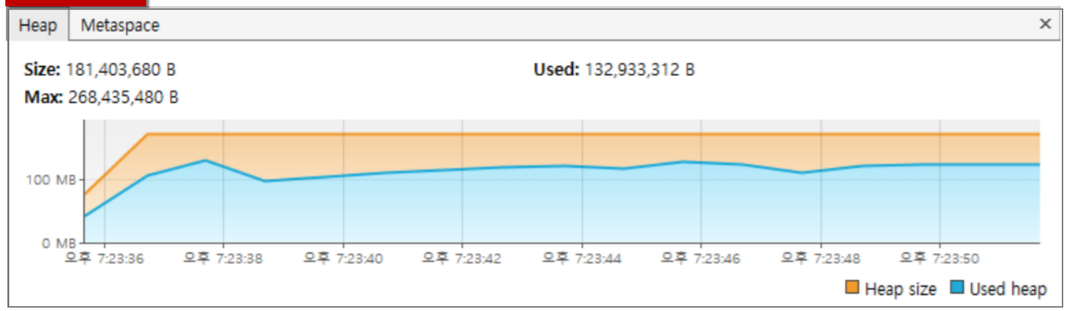
## 트레이드오프 지표

| 지표 | 개선 전 | 개선 후 | 변화율 |
|---|---|---|---|
| DB 로드 시간 | 1,727 ms | 3,553 ms | +105.7% |
| 배치 전체 시간 | 7,248 ms | 11,853 ms | +63.5% |

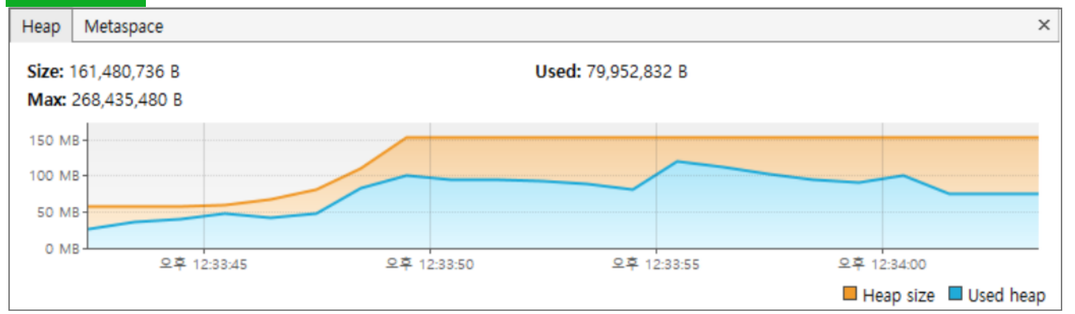
→ **OOM 발생 시 배치 전체 실패** vs 처리 시간 63% 증가 : 메모리 안정성 우선 판단

## VisualVM 힙 덤프 비교

BEFORE




AFTER



# 감사합니다