조민서

Spatial Data Backend Engineer (GIS · Digital Twin)

+82 10-5023-9977 | liging12@naver.com

블로그: https://virtualworld.tistory.com/

깃허브: https://github.com/minseojo

목차

소개

단순히 작동하는 코드를 넘어서, **명확한 목적 아래 트레이드오프를 고려하며 시스템을 만드는 일**에 흥미를 느낍니다.
기능 구현 자체보다 '**왜 이 구조를 택했는가**', '**어떤 병목이 발생할 수 있는가**', '**어떻게 확장성과 유지보수성을 확보할 것인가**', '**이 코드 한 줄이 어떤 의미를 가지는가**'와 같은 질문을 스스로에게 던지며 개발합니다.

성능 최적화, 병렬·분산 처리, 효율·안정적인 데이터 파이프라인 구축에 관심이 많으며,
시스템 전반을 깊이 있게 이해하고 설계하는 역량을 키우기 위해 실무와 학습을 병행하고 있습니다.

복잡한 문제를 단순하게 구조화하고, 명확하게 설명할 수 있는 시스템을 지향하며,
기술을 목적이 아닌 **도구로 삼아 문제의 본질에 집중하는 문제 해결형 엔지니어**를 지향합니다.

기술 스택

- **Languages**: Java, Python, C#, JavaScript, TypeScript
- **Backend**: Spring Boot (JPA, Spring MVC), REST API 설계
- **Messaging**: Apache Kafka, AWS SQS
- **Database/Storage**: MySQL, PostgreSQL, SQLite, Redis, Amazon S3, SeaweedFS (S3 호환)
- **Cloud/Infra**: AWS (EC2, RDS, S3, CloudFront, Lambda, SQS, AMI, Route 53)
- **DevOps**: GitHub Actions, Docker
- **Frontend**: React, React Native (Expo), Context API
- **etc.**: Git, GitHub, Notion, Postman, Jira, Figma

경력 요약

아이씨티웨이(주) | 정보기술연구소 | 사원 (정규직)

2024.12 ~ 재직 중

GIS 기반의 대규모 공간 데이터 처리 및 3D Tiles 마이그레이션을 수행하였고, 이를 바탕으로 현재는 GIS 버전 관리 시스템을 설계·개발하는 동시에 모놀리식 구조를 MSA 기반 분산 아키텍처로 전환하는 작업을 주도하고 있습니다.

[경력 프로젝트 1]

공간 정보 기반 3D 모델 데이터 구조 개선 및 Cesium 마이그레이션

(Explicit Tiling → Implicit Tiling)

기간: 2025.03 ~ 2025.06 | **인원**: 2인
역할 기여도: 90%
회사: 아이씨티웨이(주)
블로그 포스팅: **전 세계 3D 공간 정보를 어떻게 관리할 것인가? – Implicit Tiling, Z-Order, Bitstream, Subtree 설계/구현기**
Implicit Tiling 시각화 예시: **https://minseojo.github.io/implicit-tiling/**

프로젝트 목적

기존 GIS 엔진은 **Explicit Tiling** 방식을 사용하여, tileset.json 파일에 **모든 타일의 위치, 계층 구조, 공간 범위, 콘텐츠 경로(URI)** 를 **명시적으로 정의**하는 **정적 구조**였습니다.
이 방식은 타일 수가 많아질수록 파일 크기가 급격히 증가하며, **사용자가 지도를 조금만 움직여도 브라우저 엔진이 tileset.json의 루트부터 매번 순차적으로 탐색해야 하는 비효율**이 존재했습니다.
특히 실제 3D 모델이 존재하지 않는 희소 지역까지도 **인덱싱용 타일로 포함되어 매번 탐색 대상에 포함**되는 불필요한 노드 탐색을 했습니다.
또한, 타일 간 공간 관계나 속성 정보가 데이터베이스에 연동되지 않아, '반경 100m 내 건물 찾기' 같은 공간 질의, 특정 모델만 갱신하는 **부분 수정**, 변경된 영역만 재배포하는 **선택적 빌드**가 매우 어렵고 불가능에 가까웠습니다

기존 빌드 과정의 문제

- 모든 3D 모델의 정보(위치, 바운딩볼륨 등)을 tileset.json에 **명시적으로 나열 → 타일셋 JSON 파일 크기 증가**
- 실제 콘텐츠가 존재하지 않는 타일까지 **모든 계층에 대한 JSON 구조 생성 필요 → 불필요한 JSON 노드 탐색**
- 타일셋 구조 수정 시 → 전체 tileset.json 재생성 (전체 재 빌드) → **빌드 시간 수십~수백시간 소요**
- 중간에 빌드가 실패할 경우, **처음부터 다시 전체 재 빌드 함**
- 일부 모델만 변경되더라도 **전체 타일셋을 다시 재 빌드 해야 하기 때문에,**
- **부분 수정/재배포가 불가능하고 운영 효율이 현저히 떨어지는 구조**

이러한 문제를 해결하기 위해 본 프로젝트에서는 **3D Tiles 1.1의 Implicit Tiling 구조**를 도입했습니다.
타일을 수학적 규칙(Z-order 기반)으로 암시적으로 구성함으로써, 다음 항목을 목표로 설계했습니다.

- 타일셋 크기(용량) 감소
- 렌더링 탐색 효율 향상
- 클라이언트가 (level, x, y) 좌표만으로 타일에 랜덤 액세스 가능
- 특정 지역만 선택적으로 수정 및 배포 가능한 구조 마련
- 대규모 공간 데이터에 대한 확장성 개선

따라서 본 프로젝트에서는 지구를 일정한 규칙으로 사전에 분할하고,
그 과정에서 생성되는 **쿼드트리 탐색 구조를 Z-Order 기반의 비트스트림과 서브트리로 구현하여**,
해당 공간에 3D 모델을 배치하고 **타일 및 모델 단위의 변경 추적**과 **위치 기반 버전 비교**를 가능하게 했습니다.

기술 스택

- **GIS Engine:** C# (.NET 6)
- **Etc.**: CesiumJS, 3D Tiles 1.1, Z-order Indexing, Morton Code, Bitstream

문제 해결

1. **지구 전역을 아우르는 확장 가능한 타일링 구조 설계**
   - 기존 시스템은 특정 국가에 한정된 좌표계와 타일 체계를 사용하여, 글로벌 확장성과 유지보수에 제약이 있었습니다.
   - 이를 개선하기 위해, **WGS84 타원체 기준 0~20레벨(총 21레벨)의 전 지구를 쿼드트리로 분할하고**, 모든 타일을 (level, x, y) 좌표로 정규화된 인덱싱 체계로 표현했습니다.
   - 레벨 20 기준 타일 크기는 약 38m × 19m로, 건물 단위의 정밀한 분석이 가능하며, 글로벌 공간을 완전히 커버하면서도 특정 국가/도시만 선택적으로 처리할 수 있어 유지보수성과 확장성을 모두 확보했습니다.

   📌 ***효과**: 전 지구 대상의 통합 인덱싱 구조 확보, 국가/도시 단위의 선택적 갱신 가능*

2. **부동소수점 정밀도 손실 해결 (RTC_CENTER 도입)**
   - 고정밀 공간 해상도(레벨 20 기준 타일 크기는 약 38m × 19m)를 표현하면서, 타일 좌표와 모델 좌표 간의 미세한 부동소수점 오차가 발생했고, 이는 카메라 시점 이동 시 모델의 떨림(정합 불일치)을 야기했습니다.
   - 이를 해결하기 위해 RTC_CENTER(Root-relative to Center) 개념을 적용하여, 모델의 위치 좌표계를 각 지역 중심 기준으로 변환하고, float 연산 범위 내로 정규화 하여 GPU에서의 연산 안정성을 확보했습니다.

   📌 ***효과**:* 모델 떨림(Jittering 현상) 현상 제거, 카메라 이동 시 렌더링 안정성 확보

3. **Z-order 기반 타일 정렬 및 Bitstream 압축 구조 도입**
   - 전 지구 기준 레벨 20 타일 수는 약 1.1조 개에 달합니다.
   - 이를 압축 표현하기 위해 (level, x, y) 3차원 타일 좌표를 Z-order(Morton Order) 기반으로 정렬하여 1차원 Bitstream으로 구성했습니다.
   - Bitstream은 타일 존재 여부를 비트 단위로 압축하여, 공간적으로 인접한 타일들이 메모리상에도 인접하도록 하여 **CPU 캐시 효율 및 스트리밍 성능을 극대화**했습니다.

   📌 ***효과**:* 메모리 접근 최적화, 타일 간 공간 연관성 보존, 연속 스트리밍 효율 개선

4. **희소 데이터를 고려한 Subtree 기반 계층 구조 설계**
   - 타일 존재 여부를 비트 단위로 압축한 Bitstream 구조를 도입하여 (level, x, y) 2차원을 1차원으로 구성했습니다. 그러나 실제 3D 데이터는 대한민국 등 극히 일부분(지구 면적의 약 0.07%)에만 존재하는 **희소한 구조**입니다.
   - **따라서 전 지구를 하나의 비트스트림으로 표현할 경우 약 1.1조 비트에 달하며, 대부분의 비트가 0인 희소한 데이터임에도 불구하고 이를 매번 메모리에 로드해야 하므로, 순회 자체는 O(1)이라 하더라도 성능상 심각한 비효율이 발생합니다.**
   - 이를 해결하기 위해 전체 쿼드트리(21레벨)를 7레벨 단위로 나누어 0~6, 7~13, 14~20 레벨을 각각 하나의 Subtree로 구성했습니다. (실제 3D 모델은 16~20 레벨 사이에 존재합니다)
   - 각 Subtree는 다음 세 가지 존재 정보를 **비트스트림으로 압축하여 표현**합니다:
     - ▸ 타일 존재 여부 (5,461bit)
     - ▸ 콘텐츠(3D 모델) 존재 여부 (5,461bit)
     - ▸ 자식 Subtree 존재 여부 (16,384bit)

   📌 ***효과**:* 타일 존재 여부를 최소 단위로 분리 및 압축 → 로딩 지연 해소, 불필요한 JSON 파싱 제거

주요 성과

- **좌표 기반 Random Access 구조 구현**
  → 클라이언트는 **(level, x, y) 좌표만으로 URI 직접 접근(= 계산)→ 명시적 캐싱 가능**
  → JSON 파싱 및 DB Lookup 제거 → **O(4ⁿ) → O(1)**
  → 타일 탐색 비용 제거, 트래픽 감소, 렌더링 속도 개선
  → **중앙 서버 없는 독립형** 클라이언트 뷰어 가능
  → 오프라인 캐시 및 저장 재활용 가능
  → 전 지구 대상에서도 **일관된 타일 주소 구조 유지**

- **3D 모델의 공간정보화 및 분석 가능 구조로 전환**
  → 모델을 단순히 렌더링 바이너리 데이터가 아닌 "타일+좌표안의 3D 데이터(Feature{geom 등})" 기반으로 관리
  → 반경 기반 질의, 주소 필터링, 변경 이력 추적 등 가능

- **타일셋 메타데이터 경량화**
  → 기존 JSON 대비 파일 크기 약 30~40% 감소 → 캐싱 효율 증가
  → 초기 로딩 시간 단축, 네트워크 대역폭 절감

- **렌더링 성능 향상**
  → Z-order 기반 메모리 정렬
  → 트리 탐색 제거 → FPS 20~30% 향상
  → 대규모 공간 데이터에서도 줌/팬 반응성 개선

- **후속 시스템(버전 관리 등) 연계 기반 마련**
  → 좌표 기반 구조는 Git-like 시스템에서 변경 추적 핵심 요소로 활용

주요 키워드

지구 전체 그리드 분할, RTC Center, Z-order 인덱싱, Bitstream 압축, Subtree 구조, 희소 데이터 최적화, 트리 탐색 O(1) 개선, 랜덤 액세스, tileset.json, 자료구조 설계/구현, 공간 정보 버전관리 시스템 기반 구축

관련 포스팅

- **[Blog] 전 세계 3D 공간 정보를 어떻게 관리할 것인가? – Implicit Tiling, Z-Order, Bitstream, Subtree 구현기**
- **[Blog] 공간 데이터 성능 최적화: Z-order Indexing (feat. 캐시, 압축, I/O)**
- **[Blog] 컴퓨터는 실수를 어떻게 계산할까? 부동 소수점 표현의 원리와 한계**
- **[Blog] 컴퓨터에 지구를 위치 시키는 법**

마이그레이션 적용한 Cesium 오픈소스 깃허브

- **CesiumGS/3d-tiles/ImplcitTiling**

[경력 프로젝트 2] 공간 정보 버전관리 시스템 설계/개발 (모놀리식 → MSA 전환)

기간: 2025.06 ~ 진행 중 | **인원**: 2인
역할 기여도: 50% / 타일 빌드 Saga 흐름 및 MSA 아키텍처 구현 전담
주관 기관: 아이씨티웨이(주).

프로젝트 목적

기존 모놀리식 시스템에서는 전체 타일셋을 재 빌드하는 데 수십 시간에서 수백 시간이 소요되었으며,
빌드 도중 네트워크 오류나 OOM(Out of Memory) 등의 문제가 발생하면 처음부터 다시 빌드를 수행해야 했습니다.
왜냐하면 기존 Explicit Tiling 방식은 tileset.json 파일에 모든 타일의 계층 구조와 위치 정보를 명시적으로 나열하는 구조로, **일부 지역만 수정되더라도 해당 계층 전체를 tileset.json 파일에 재정의해야 했기 때문입니다.**

이로 **인해 부분 빌드나 병렬 처리가 불가능했고**, **운영 부담과 시간 낭비를 야기했습니다**.
이러한 비효율을 해소하기 위해, 경력 프로젝트 1에서는 타일 표현 방식을 Explicit Tiling에서 Implicit Tiling으로 전환하여 **구조의 일관성**과 **확장성을 확보**하였고,

본 프로젝트에서는 도메인별 기능을 분리하고 메시지 기반의 MSA 구조로 재설계하여 빌드 병렬화, 장애 격리, 선택적 처리 기반의 유연한 아키텍처를 구현하고 있습니다.

기술 스택

- **Spring Boot** ***기반 REST API 서버*****:** 타일 빌드 요청, 커밋/체크아웃 등 주요 기능을 REST API 형태로 제공
- **Apache Kafka** ***기반 비동기 메시징 처리 구조***
  → RabbitMQ 등 전통적인 메시지 브로커는 1회성 전달에 초점이 맞춰져 있어 메시지 유실 시 복구가 어렵지만, Kafka는 이벤트 로그를 저장하여 다수의 서비스가 동일 이벤트를 독립적으로 구독·재처리할 수 있고**,**
  메시지 손실 없이 정합성을 유지할 수 있어 타일 빌드와 같이 후속 단계가 중요한 작업에 적합하다고 판단
  → 또한, 타일 빌드 시 **하드웨어를 수평 확장(Scale-out),** Kafka의 파티션과 컨슈머 그룹도 함께 확장하여 **병렬 처리 성능을 선형적으로 향상시킬** 수 있어, 대규모 지역 타일링에서도 **안정성과 처리 속도를 확보 가능**

- ***MSA 아키텍처 + Saga 패턴(Choreography)***
  → 타일 빌드는 **소스 이미지 수집 → Geometry 계산 → 텍스처 병합 → 3D 모델 생성(Geometry + Texture) → 타일링(Subtree 구성)**의 순차적인 단계로 구성됩니다.
  → 이 과정을 하나의 중앙 Orchestrator가 관리하는 방식도 가능하지만, **서비스 간 결합도가 높아지고, 조정 로직이 비대해지는 문제**가 있었습니다.
  → 각 빌드 단계는 입력과 출력이 명확하게 정의되어 있어, 이전 단계의 결과만 확보되면 **별도의 상태 공유 없이 독립적으로 실행이 가능**하므로, **비동기 이벤트 기반 처리에 적합**했습니다.
  → 이에 따라 각 단계가 Kafka 이벤트를 기반으로 자체적으로 다음 단계를 트리거하는 **Choreography 방식**을 적용해, **모듈 간 느슨한 결합과 독립 배포, 장애 격리, 병렬 확장성**을 확보했습니다.

- **PostgreSQL + S3 호환 스토리지 (SeaweedFS)**
  → 타일, 3D 모델, Feature 메타데이터는 PostgreSQL에 저장하고, 대용량 GLB 및 서브트리(Subtree: 타일 계층 구조 정의) 파일은 S3 호환 스토리지에 분리 저장하여, 역할 책임을 명확히 하고 효율적인 접근 구조 구현

- ***C# 기반 타일 빌드 엔진 연동***
  → Java에서 Kafka를 통해 빌드 요청을 발행하면, C# 워커 스레드가 이를 수신해 타일 빌드 모듈을 실행하고, 생성된 결과(GLB/Subtree 등)는 Kafka를 통해 다시 전달되어 S3에 청크 단위로 병렬 업로드합니다..

담당 문제 해결

1. **단일 트랜잭션 기반 타일 빌드의 복잡성과 비효율성 → Saga 기반 MSA 전환**

- 기존에는 타일 빌드 전체 과정을 하나의 단일 트랜잭션 또는 일괄 처리로 수행했으나, 단계별 실패 복구가 불가능하고, 처리 단위도 커 확장에 불리
- 이를 Kafka 기반 Saga 패턴으로 재설계하여, 타일 빌드 과정을 **소스 수집 → Geometry 계산 → 텍스처 병합 → 3D 모델 생성 → 타일 배치** 로 세분화(모듈)
- 각 단계는 서로 다른 마이크로서비스로 분리하여 Kafka 이벤트를 통해 비동기 전파되며, 중앙 오케스트레이터 없이 **코레오그래피 방식**으로 구성

📌 ***효과:*** 서비스 간 결합도 최소화 및 유연한 장애 복구/재시도 구조 확보

2. **타일 ID 구조 단순화 및 파티셔닝 전략 적용 → 효율적인 빌드 분산 처리 기반 마련**

- **기존 타일 ID(PK)는 level_x_y 문자열 형태로 구성되어 있었으며, 이는 다음과 같은 병목과 비효율을 야기**
  • 문자열 비교 기반 정렬은 비용이 크고,
  • PostgreSQL에서 WHERE IN (...) 조건문은 범위 기반 인덱스를 활용하지 못해 성능 저하
  • 파티셔닝 기준이 명확하지 않아 **병렬 처리 시 작업 청크 분리가 어려움**
- 이를 해결하기 위해 타일 후보 테이블의 PK를 문자열 형태의 level_x_y 대신 **Auto-Increment 정수형 PK**를 부여하고, 해당 PK를 기준으로 **Range 파티셔닝을 적용**
  → 타일 ID는 정렬이나 순서 의미가 없기 때문에, 단순 정수형 ID만으로도 충분
  → WHERE id BETWEEN a AND b 형태의 Range Scan을 사용하면 인덱스 효율을 극대화할 수 있음
  → 병렬 처리를 위한 Task 단위를 정수형 PK 범위 기반 청크로 나눔
- **청크 단위 빌드 Task를 Kafka 메시지로 생성**하고, 각 메시지에 포함된 ID(PK) 범위만 처리하도록 설계
  → Consumer는 해당 ID 범위에 해당하는 **독립성이 보장되는 타일만 병렬 처리하는 단순한 구조**
  → 실제 타일은 약 19m × 38m 크기로 세밀하게 분할되어 있어, 개별 처리 비용이 작고 배치 처리에 적합
  → Task 생성 시 각 타일 빌드에 적용되는 **텍스처, Feature 밀집도 등 분석을 통해 빌드 시간 편차를 줄이는 Task 배치 전략**도 도입 예정
- **Kafka 브로커는 라운드로빈 파티셔닝 방식을 채택**
  → **타일 빌드 순서에 의존하지 않게 설계 했기 때문에**, 키 파티셔닝보다 라운드로빈이 더 효과적
  → Consumer 간 **처리량 분산이 단순하게 이루어져 병목 없이 확장 가능**

☑ **이런 구조가 가능한 이유: 3D 모델이 Implicit Tiling 주소 체계에 따라 사전 배치되어 있음**

- 빌드 이전에 모든 3D 모델은 중심 위경도(lon/lat)를 기반으로 **Implicit Tiling의 (level, x, y) 좌표로 매핑 가능**
  → 즉, 어떤 모델이 어떤 타일에 속하는지는 빌드 전에 완전히 결정 가능
  → 이 과정에서 각 타일에 필요한 **Feature 정보(위치, 바운딩볼륨, 텍스처 ID 등)는 PostgreSQL에 저장**, 텍스처 이미지 파일은 S3 호환 스토리지(SeaweedFS)에 저장되어 있음
  → 빌드 시에는 정수형 PK 기반으로 Feature 정보를 **청크 단위로 Range Scan**하여 효율적으로 조회 가능하며,
  → 텍스처 이미지는 S3의 HTTP 기반 API를 통해 **독립적인 비동기 방식으로 병렬 요청** 가능
  → 이처럼 메타데이터와 파일을 분리 저장함으로써, 각 타일 빌드 Task는 DB와 S3에서 필요한 정보를 **의존성 없이 동시에 비동기 처리**할 수 있음

📌 ***효과:*** **병렬 처리 효율 향상 및 확장성 확보**

- 기존 level_x_y 문자열 기반 타일 ID 비교 대신,
  • **정수형 PK 기반 Index Range Scan + Range 파티셔닝 + Kafka 라운드로빈 분산** 조합으로 전환
  → 병렬 처리 단위가 명확해지고, **청크 단위 배치 처리에 최적화된 구조 확보**
- **Kafka 메시지 생성 → 병렬 Task 처리 → 타일 빌드 완료**까지의 전체 흐름은
  → 타일 간 의존성이 없기 때문에 순서에 영향을 받지 않고 **완전히 병렬**로 수행 가능
  → 각 메시지는 partitionTable, rangeStartTileId, rangeEndTileId 등의 정보를 포함해
  각 Consumer가 **특정 ID 범위만을 독립적으로 처리**하도록 구성됨
  → PostgreSQL에서 **Index Range Scan**으로 빠르게 후보 타일 조회
  → 텍스처 등 부가 데이터는 S3(SeaweedFS)를 통해 **랜덤 액세스 + 비동기 호출**
- 이 구조를 통해,
  → **정수형 PK 기반 쿼리는 문자열 IN 절보다 수십 배 빠른 성능** 제공
  → 빌드 요청 간 순서/의존성이 없으므로 **선형 확장성 및 고가용성 처리** 가능
  → 타일 단위로 분리된 **독립적 재빌드**도 용이

3. 데이터 규모 및 인프라 증가 대응 → 선형 확장 가능한 구조 확보

- Kafka 파티션 수 및 Consumer Group 수를 조절해 **서버 수 증가에 따라 처리량을 선형 확장 가능한 구조로 설계**
- **DB를 수평 확장한 경우에도**, **Kafka 메시지**에 접속 정보 + 파티션 테이블명만 포함하면 각 Consumer가 독립적으로 해당 파티션의 타일 처리 가능
- PostgreSQL + S3 호환 스토리지(SeaweedFS) 분리 구조로, 메타데이터/파일 저장을 분리하고,
  3D 모델 렌더링 시 **DB(lookup 테이블) 연결 없이도 S3만으로 모델 접근 가능**

4. 이벤트 손실 없는 메시지 전파 → Outbox + Polling 기반 이벤트 발행 구조 설계

- Kafka 메시지 전송 실패나 롤백 발생 시 메시지 유실 위험을 제거하기 위해, PostgreSQL에 Outbox 테이블 도입
- 도메인 트랜잭션 내에서 Outbox 테이블에 이벤트를 먼저 기록하고, 별도 폴링 쓰레드가 PENDING 상태의 이벤트만 Kafka로 발행
- 발행 성공 시 SENT로 상태 변경, 실패 시 재시도 가능

📌 ***효과:*** CDC(Debezium) 없이도 메시지 손실 없이 정합성을 보장할 수 있는 안전한 이벤트 흐름 구축

5. 중복 처리, 쿼리 병목, 전체 재빌드 → 불필요한 처리 제거 및 효율 최적화

- SHA-1 해시 기반 중복 체크로 동일한 소스이미지/3D모델/Feature/Subtree는 재사용하고,
  변경된 데이터만 선별 빌드 → **재처리 비용 최소화**

성과 요약

1. Saga 기반 MSA 전환으로 확장성과 장애 대응성 확보

- 기존 단일 트랜잭션 방식의 빌드 과정을 **Kafka 기반 Saga 패턴으로 재설계**
- 빌드 과정을 단계별 모듈(소스 수집 → Geometry 계산 → 텍스처 병합 → 3D 모델 생성 → 타일 배치)로 분리
- **중앙 오케스트레이터 없이 코레오그래피 방식으로 설계**, 각 마이크로서비스 간 느슨한 결합 구조
  📌 **단계별 재시도 및 장애 복구가 가능해졌으며**, 유지보수성과 서비스 탄력성이 향상됨

2. 타일 ID 및 파티셔닝 전략 개선을 통한 빌드 병렬화 구조 설계

- 기존 문자열 기반 ID(level_x_y)의 성능/확장성 한계를 해결하기 위해
  → **Auto-Increment 정수형 PK 부여 + Range 기반 파티셔닝 적용**
- 각 Kafka 메시지에는 rangeStartId, rangeEndId 정보만 포함 → Consumer는 해당 범위만 병렬 처리
- 병렬 처리 단위가 정수 범위 기반으로 단순/명확해졌고, **청크 기반 배치 처리 최적화 가능**
  📌 **쿼리 성능 수십 배 향상** (IN → Range Scan), **정렬/작업 분리 부담 제거**

3. **타일 간 독립성 보장을 통한 선형 확장 구조 확보**

- 타일 간 의존성이 없도록 설계해, Kafka 라운드로빈 분산 구조에서도 **순서 제약 없이 완전 병렬 처리 가능**
- 타일 빌드에 필요한 모든 정보(Feature 메타데이터, 텍스처)는 DB와 S3에 사전 저장 → 빌드 시점에 독립 조회 가능
- 텍스처는 S3 API로 비동기 처리 → DB와 S3를 병렬 액세스

  📌 **노드 수 증가에 따라 처리량이 선형 확장 가능, 전체 타일 재빌드 없이도 부분 재처리 가능**

4. **Outbox + Polling 기반 안전한 이벤트 발행 구조 설계**

- Kafka 전송 실패로 인한 메시지 유실 방지를 위해 PostgreSQL Outbox 테이블 도입
- 트랜잭션 커밋 이후, Polling Thread가 PENDING 상태 메시지를 Kafka로 전송
- 전송 성공 시 SENT 상태 변경, 실패 시 재시도 가능

  📌 **CDC 없이도 데이터 정합성을 유지하며 메시지 손실 없이 안정적으로 전파 가능**

5. **중복 처리 제거 및 재빌드 최소화**

- SHA-1 해시 기반 중복 검사를 도입 → 소스 이미지, 3D 모델, Feature, Subtree 단위로 **캐싱/재사용**
- 변경 감지 기반으로 타일 빌드 수행 → **불필요한 재처리 최소화, 리소스 낭비 방지**

  📌 **빌드 효율 극대화 및 전체 재처리 시간 수십 배 절감**

주요 키워드

Saga 전환, Kafka 이벤트, 정수 PK, Range 파티셔닝, 라운드로빈 분산, 청크 처리, Outbox 발행, 비동기 빌드, 타일 독립성, 사전 매핑, DB+S3 분리, 해시 중복 제거, Range 쿼리, 무순서 병렬, CDC 없이 안전

관련 포스팅

- [Blog] Kafka를 이해하려면 로그부터 이해하자! (feat. Append-Only, Write Ahead Log, Event Sourcing)
- [Blog] Kafka는 왜 빠를까? 운영체제의 Zero-Copy (feat. DMA, snedfile(), Page Cache)
- [Blog] Redis BGSAVE가 OOM을 발생시키는 이유 (feat. fork, Copy-on-Write)
- [Blog] UNIX 설계의 핵심 – File, System Call, inode, Mount (The UNIX Time-Sharing System, 1974)

[경력 프로젝트 3] 3D 파일 포맷 변환 파이프라인 개발 (B3DM → glTF 2.0)

기간: 2024.05 ~ 2024.06 | **인원**: 2인
역할 기여도: 70% / B3DM → glTF 2.0 변환 로직 설계 및 C# 기반 직접 구현 (바이너리 파싱, 메타데이터 저장 구조 변경 등)
주관 기관: 아이씨티웨이(주)

프로젝트 목적

- **Implicit Tiling 도입**에 따라, 레거시 B3DM 파일을 표준 glTF 2.0 포맷으로 마이그레이션 필요
- Cesium 공식 오픈소스 기반의 기존 npm 라이브러리는(3d-tiles-tools) 다음과 같은 한계 존재
  - 모든 파일에 일괄 **Draco 압축** 및 Cesium 자체에서 판단한 **불필요한 리소스 제거**
  - 이로 인해 **텍스처 손상 및 비즈니스용 메타데이터 손실 발생**
  - **Node.js 기반 단일 스레드 I/O 처리 구조로** 인해 변환 속도 저하
    → 약 1000개 파일 변환에도 약 **20분 소요되는 병목** 존재

→ 품질 손실 없이, **빠르고 유연한 변환 도구의 직접 구현 필요성 대두**

기술 스택

- **C#: (바이너리 파싱 및 직렬화)**
- **3D Tiles:** B3DM, glTF 2.0

주요 구현

- **C# 기반으로 B3DM 파서 직접 구현**
  → RTC_CENTER, Feature Table, Batch Table 파싱 및 glTF 2.0 재직렬화
- 좌표계 변환 (Y-up → Z-up), glTF 메타데이터 통합 구조 설계
- 테스트 코드 구현

성과

- 변환 속도 **20분 작업 → 5초로 단축 (약 240배 향상)**
- 텍스처 및 메타데이터 **손상 없이 보존**, 테스트 코드로 **신뢰성 확보**
- glTF 2.0 기반 체계로 **관리성과 확장성 확보**
- **테스트 코드**로 신뢰성 확보

주요 키워드

B3DM, glTF 2.0, 바이너리 파싱, 직렬화/역직렬화, 바이너리 파일 포맷 마이그레이션, 좌표계 변환, 데이터 무결성 검증, 텍스처 품질 보존

관련 포스팅

- **[Blog] 3D Tiles 1.0 주요 포맷 정리: B3DM**
- **[Blog] glTF 2.0 구조 및 구성 요소**

마이그레이션 적용한 Cesium 오픈소스 깃허브

- **https://github.com/CesiumGS/3d-tiles/tree/main/specification/TileFormats/glTF**
  → MIGRATION.adoc

[팀 프로젝트 1] Text-to-Image 생성형 AI 기반 네 컷 만화 생성 앱

기간: 2024.04 ~ 2024.12 | **인원**: 3인
역할 기여도: 백엔드 70%, 프론트 25%, AI 30% / 시스템 설계 및 백엔드 아키텍처 총괄
주관 기관: 과학기술정보통신부 (SW 마에스트로 15기)
깃허브: 해당 프로젝트는 레포지토리가 비공개 상태입니다.

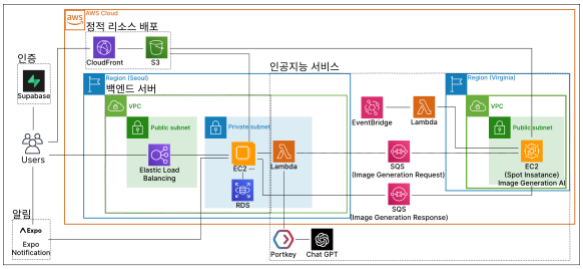

서비스 대표 캐릭터 (말랑)

프로젝트 목적

사용자 일기 내용을 바탕으로 감정과 개성을 반영한 AI 이미지를 생성하고,
이를 동일한 캐릭터 스타일로 구성된 네 컷 만화로 시각화하여 개인화된 콘텐츠를 제공하는 모바일 앱 개발.
기존의 DALLE·Midjourney 등 이미지 생성 모델은 캐릭터 일관성 유지가 어려워, 외주 제작한 약 80장의 캐릭터 이미지로 LoRA 기반 Stable Diffusion 모델을 파인튜닝함으로써 귀여운 동물 캐릭터의 일관성을 유지함.

기술 스택

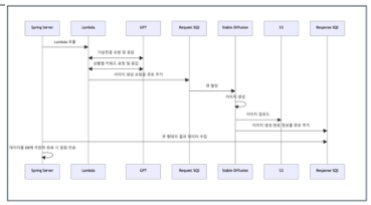
- **Backend**: Java, Spring Boot (RESTful API 서버 구현 및 비즈니스 로직 처리)
  MySQL, JPA (도메인 기반 데이터 모델링 및 쿼리 최적화)
- **GPU 이미지 생성 서버 (Python 기반)**
  (GPU 서버는 EC2 Spot 인스턴스로 구성, 이미지 생성 시에만 일시적으로 가동)
  Python: Stable Diffusion 기반 이미지 생성
  AWS SQS + Python 워커 스레드: SQS 메시지 폴링 → Stable Diffusion 실행 → S3 업로드
- **서버리스 처리 (Python Lambda)**
  AWS Lambda (Python): 이미지 생성 요청을 수신해 SQS에 메시지를 발행하는 서버리스 트리거 함수로 사용
- **Cloud(AWS) & Infrastructure**
  EC2 On-Demand: Spring Boot 백엔드 서버 운영
  EC2 Spot: Stable Diffusion 전용 GPU 이미지 생성 서버
  RDS for MySQL: 운영용 관계형 데이터베이스
  AWS S3: 생성 이미지 및 정적 리소스 저장소
  CloudFront + Presigned URL: 이미지 캐싱 및 보안 전달
  EC2 Auto Scaling Group + ELB: 서버 자동 확장 및 부하 분산
  Docker: 개발 및 배포 환경 컨테이너화



인프라 시스템 구성도

- **DevOps / 알림 / 모니터링**:
  GitHub Actions: CI/CD 파이프라인 자동화 및 무중단 Rolling 배포
  Firebase Cloud Messaging (FCM): 이미지 생성 완료 실시간 사용자 알림 전송
  AWS CloudWatch: Lambda, EC2, SQS 로그 및 지표 모니터링, GPU Spot 인스턴스 트리거 및 상태 추적
  Sentry: 백엔드 및 프론트엔드 에러 실시간 모니터링
  Amplitude: 사용자 행동 로그 수집 및 사용 패턴 분석
- **AI / Personalization**
  Stable Diffusion: 텍스트 기반 이미지 생성
  LoRA Fine-tuning: 사용자 감정·개성을 반영한 이미지 생성 모델
- **Frontend**: TypeScript, React Native(앱 개발), Expo (앱 빌드 및 배포)
- **협업 도구:** Postman (API 테스트 및 문서화), Git & GitHub (git-flow 기반 브랜치 전략 및 PR 리뷰),
  Notion (회의록 및 스프린트 관리), Discord, Figma, 1Password (공용 비밀번호 및 시크릿 키 관리)

주요 담당 역할 및 성과


이미지 생성 시퀀스 다이어그램

- **AI 이미지 생성 아키텍처 설계 및 분리 운영 체계 구축**
  Stable Diffusion 기반 이미지 생성을 기존 백엔드와 완전히 분리하기 위해, Lambda(Python)를 트리거로 사용하고, 이미지 요청은 AWS SQS를 통해 큐잉 후 GPU Spot 인스턴스에서 Python 워커 스레드가 비동기 처리하는 구조로 설계
  → AI 파이프라인과 백엔드 서버를 분리함으로써, 서로 다른 리소스 요구(CPU vs GPU)를 고려한 인프라 운영이 가능해졌고, 단일 서버에서 발생할 수 있는 부하 집중과 코드 충돌 문제를 해소
  → GPU 서버는 필요 시에만 일시적으로 실행되는 Spot 인스턴스를 활용하여 **AI 인프라 비용 56% 절감**
  → 백엔드는 CPU 위주 EC2에서 안정적으로 운영되며, AI 서버 변경 시 백엔드 코드에 영향을 주지 않아 **유지보수성과 확장성 모두 확보**


Diary 서비스가 의존하는 하위 서비스

- **도메인 중심 설계(DDD) 기반 구조 설계 (안정적인 트랜잭션)**
  핵심 도메인 '일기(Diary)'는 캐릭터, 이미지, 번역 등 다양한 하위 도메인에 의존하는 핵심 애그리거트로, DiaryService는 여러 하위 서비스(DiaryCharacterService, DiaryS3Service, DiaryPaintingImageCloudFrontService 등)를 의존하는 통합 서비스로 설계했습니다.
  → 도메인 중심 설계를 적용하면서도 의존성 관리를 위해 각 기능별 역할을 명확히 나눴고, 트랜잭션 단위의 책임을 분리하여 **가독성과 테스트 용이성을 확보**했습니다.
  → 단일 도메인 서비스가 여러 인프라 서비스(S3, CloudFront, Lambda 등)와 통합되어 있음에도 불구하고, 단일 트랜잭션 흐름에서 부작용 없이 작동하도록 **@Transactional 기반의 안정적인 파이프라인을 구성**했습니다.

- **RESTful API 설계 및 DTO 책임 분리 구조 구성**
  RESTful한 URL 자원 표현, HTTP 메서드 분리, 명확한 응답 코드 등의 표준을 준수하면서,
  컨트롤러 계층에서는 도메인 객체를 직접 노출하지 않고 RequestDTO.from() / ResponseDTO.of()와 같은 팩토리 메서드를 통해 DTO로 변환하는 구조를 설계했습니다.
  → 이를 통해 **응답 구조의 일관성과 명확성을 유지하고**, 코드 재사용성과 **유지보수성을 높였습니다.**
  → 또한, Postman을 활용해 명세 문서를 병행 관리하여, **API 소비자(팀원)와의 커뮤니케이션을 체계화**하고 안정적인 API 운영 기반을 마련했습니다.
  → 서비스 계층에서는 JPA의 Lazy Loading이 트랜잭션 범위 내에서만 안전하게 작동함을 고려해 도메인 객체를 반환하도록 하여, **데이터 정합성을 유지**하면서도 복잡한 비즈니스 로직 내에서의 연관 데이터 접근을 효율적으로 처리할 수 있도록 구성했습니다.

- **반정규화를 통한 조회 성능 개선**
  사용자가 무한 스크롤로 콘텐츠를 조회할 때, 이미지·해시태그 테이블과의 조인으로 인한 성능 저하(N+1 문제)가 발생했습니다. 이를 해결하기 위해 Diary에 포함되는 4장의 이미지, 해시태그들을 **반정규화(Flat 구조)**하여 단일 테이블 조회가 가능하도록 개선하였습니다.
  → 일기가 변경되면 4장의 이미지, 해시태그도 항상 전체 교체되므로 데이터 중복에 따른 정합성 문제는 제한적이며, 그 결과 **응답 속도가 약 46% 향상**되어 사용자 체감 성능을 개선했습니다

- **Presigned URL을 활용한 이미지 접근 구조 개선 및 보안 정책 수립**
  무한 스크롤 시마다 백엔드를 경유해 S3에 접근하면서 **발생한 트래픽 과부하와 비용 문제**를 해결하기 위해, **클라이언트가 프라이빗 이미지에 직접 접근할 수 있도록** Presigned URL 기반 구조를 설계했습니다.
  → CloudFront를 연동해 캐싱 효과를 극대화하고 서버 부담을 줄였습니다.
  → 처음에는 **사용자별 디렉터리 + Signed URL을 고려**했으나, **향후 공개 일기 공유 기능 도입 시 Signed URL 방식은 다른 사용자의 비공개 이미지까지 접근 가능한 보안 이슈가 있어**, 서버에서 일기 공개 여부를 검증 후 **일기 한 건, 한 건에 대해 Presigned URL을 발급**하는 방식으로 보안성과 유연성을 확보했습니다

- **다국어 지원 및 번역 시스템 구현 (정적 + 동적 콘텐츠 분리 설계)**
  클라이언트의 Accept-Language 헤더 기반으로 언어를 자동 설정하는 Spring의 LocaleResolver와 커스텀 MessageSource를 활용하여, **버튼 텍스트, 안내 메시지 등 정적 콘텐츠**는 .properties, .xml 기반의 프로퍼티 파일로 관리하고, 언어에 따라 자동으로 적용되도록 구성

  한편, **캐릭터 이름, 캐릭터 설명 등 DB 기반 동적 콘텐츠**는 @TranslatableField, @TranslateMethodReturn 커스텀 애노테이션과 AOP를 활용한 **자동 번역 시스템**을 설계하여, **서비스 반환 시점에 프록시가 리플렉션으로 번역 대상 필드를 탐색하고, 언어별 번역 테이블에서 값을 조회하여 동적으로 주입**되도록 구현

  → 해당 구조는 새로운 언어를 도입할 때 **언어별 번역 테이블만 추가하면 전체 시스템에 자동 반영**되도록 설계되어, 별도 로직 수정 없이 빠른 언어 확장이 가능합니다.
  또한, **도메인 내부의 Lazy 필드도 번역 시점에 Eager Loading하여 필요한 데이터만 즉시 로딩**하고, **계층적으로 중첩된 필드도 재귀적으로 탐색 및 변환**되도록 구성함으로써 유지보수성과 확장성을 동시에 확보했습니다.

  관련 포스팅: *LocaleResolver* / *MessageSource (Properties, XML, YAML)* / *프록시 객체 vs 일반 객체*

- **실시간 데이터 상태 제공 (FCM 기반 알림 시스템)**  ← 알림 이미지
  AI 이미지 생성은 GPU 서버에서 처리되며, **비용 절감을 위해 GPU 서버를 필요 시에만 실행되는 Sp ot 인스턴스를 활용함으로써 일정 수준의 생성 지연 (최대 12시간)**이 존재합니다.
  → 이를 고려하여 **Firebase Cloud Messaging (FCM)** 을 통해 이미지 생성 완료 시점을 사용자에게 실시간으로 안내함으로써, 지연되는 프로세스에서도 명확한 상태 인지를 가능하게 했습니다.
  → 캐시 무효화와 실시간 알림 시스템을 통해 AI 이미지 생성의 비동기성과 지연 상황에서도 사용자 신뢰성과 만족도를 유지하는 UX를 구현했습니다.

- **캐시 무효화 및 UX 개선** 플레이스홀더 → 
  → 일기 수정 시 기존에 캐싱된 이미지 데이터를 무효화하여 최신 생성 결과가 반영되도록 처리했습니다.
  → 이미지 생성이 완료되지 않은 경우, 상태값(generating)에 따라 플레이스홀더 이미지를 우선 노출하여 UX 흐름이 끊기지 않도록 구성했습니다.
  → 또한, **React Native의 Context API**를 활용해 **최신 10건의 일기(내용 + 이미지)**를 로컬 스토리지에 캐싱함으로써 **오프라인 모드**를 지원하고, 네트워크 연결이 불안정한 상황에서도 안정적인 사용자 경험을 제공했습니다.

- **AWS AMI 기반 Auto Scaling 구조 설계**
  Spring Boot 백엔드 서버를 **EC2 AMI로 미리 빌드 및 패키징**하여, Auto Scaling 시 빠르게 신규 인스턴스를 배포할 수 있는 구조로 설계했습니다.
  → 이는 EC2 Auto Scaling Group과 연동되어, 트래픽 증가 시 **즉시 신규 인스턴스를 AMI로부터 생성해 자동 확장**이 가능하도록 하였고,
  → 이를 통해 **무중단 서비스 유지와 배포 속도 안정성을 동시에 확보**

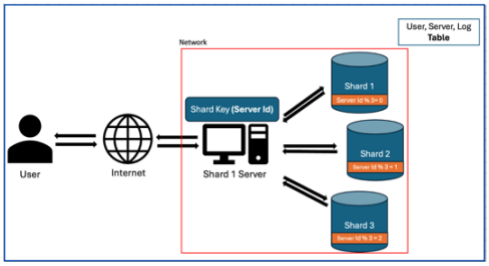
- **무중단 배포를 위한 CI/CD 파이프라인 구축**
  → GitHub Actions 기반으로 **CI/CD 파이프라인을 구축하고, AMI 기반 Rolling 배포 방식을 적용**하여 배포 자동화 및 안정성을 확보했습니다.
  → 그 결과, 배포 소요 시간이 **약 70% 단축**되었고, 운영 중 발생한 배포 중단 사고는 **0건을 유지**

주요 키워드

AI 인프라 비용 절감, 서버 부하 분산, 트랜잭션 관리, N+1, 비공개 이미지 보안, 무중단 배포 자동화, 다국어 지원, 응답 구조 일관성 유지, 컨트롤러-서비스 책임 분리, 지연 상황에서 UX 개선

[팀 프로젝트 2] MySQL 기반 Modular Sharding 알고리즘 성능 연구

기간: 2024.03 ~ 2024.06 | 인원: 4인
역할 기여도: 35%, 팀장, 샤딩 환경 구축, 성능 테스트, 모니터링 웹 구현
주관 기관: 전북대학교 (캡스톤)
실행 영상: https://www.youtube.com/watch?v=zAhyplsdYAA
깃허브: https://github.com/minseojo/modular-sharding



시스템 구성도

기술 스택

- **Backend**: Java, Spring Boot
- **DB 및 데이터 처리**: MySQL (약 960만 건), JPA, JDBC Batch, 멀티코어 기반 병렬 처리
- **성능 최적화**: JVM Heap 설정, MySQL 패킷 크기 조정, 인덱스 튜닝 (풀 스캔 / 레인지 스캔 / 커버링 인덱스)
- **웹 시각화**: Spring MVC, Thymeleaf
- **성능 분석 도구**: Apache JMeter (샤드 수별 쿼리 부하 테스트)

주요 담당 역할

- **모듈러 샤딩 알고리즘 설계 및 적용**: 960만 건 이상의 데이터를 샤드 수(1~3개)별로 분할하여 성능 비교 실험
- **쿼리 성능 실험 설계**: 풀 스캔, 인덱스 레인지 스캔, 커버링 인덱스 기반의 성능 차이를 체계적으로 측정
- **접근 방식별 성능 비교**: 순차 vs. 랜덤 탐색에서 샤딩 유무에 따른 정량적 성능 분석
- **멀티코어 기반 병렬 처리 구조 설계**: 샤드 수 증가에 따른 성능 향상을 실험적으로 입증

성능 분석 결과

- **테이블 풀 스캔:** 샤드 수 증가에 따라 선형적인 병렬 처리 효과 확인
  → 샤드 수 증가에 따라 선형적인 병렬 처리 효과 확인
- **인덱스 레인지 스캔**: 샤드를 1개에서 3개로 증가 시 최대 2배 성능 향상
- **커버링 인덱스**: 동일 조건에서 최대 1.5배 성능 향상
- **인덱스 비교**: 커버링 인덱스는 인덱스 레인지 스캔 대비 최소 2.6배 ~ 최대 5.5배까지 우수한 성능

문제 해결 사례

- JDBC 배치 처리와 트랜잭션 최적화로 VDI 환경에서 **삽입 시간 11시간 → 4시간 단축**
- MySQL 패킷 크기 조정을 통해 대용량 조회 시 발생하던 **전송 오류 해결**
- JVM 메모리 튜닝 및 레코드 구조 변경을 통해 **OOM(Out of Memory) 문제 해결**

성과 및 시사점

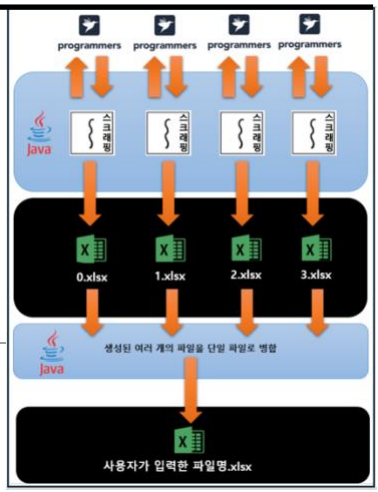
- 샤딩은 테이블 풀 스캔 환경에서 **의미 있는 병렬 처리 성능 개선 효과**가 있음을 실험적으로 입증
- **샤드 키 설계**와 **병렬 처리 전략**이 대규모 시스템 성능에 결정적인 요소임을 확인
- 실험 데이터를 기반으로 샤딩 전후의 쿼리 성능을 정량 비교하여, **대규모 DB 설계에 대한 실무 기준**을 도출

주요 키워드

데이터 병목 해소, 대용량 처리 가속화, 병렬 처리 구조 설계, 인덱스 효율 최적화, 패킷 오류 해결, 멀티코어 활용 극대화, 메모리 병목 제거, 쿼리 정량 분석

[개인 프로젝트 1] 병렬 채용 정보 스크래퍼 개발

기간: 2023.09 ~ 2023.11 | 인원: 1인(개인)
역할 기여도: 100%
실행 영상: https://www.youtube.com/watch?v=a0PJ3KzdYwk
깃허브: https://github.com/minseojo/job-informaton-scraper



시스템 구성도

기술 스택

- **언어 및 프레임워크**: Java, Selenium (4.6+ 내장 WebDriver)
- **멀티스레딩 및 동시성 처리**: Java Multi-threading, Thread Pool
- **파일 처리 및 출력**: Apache POI (엑셀 파일 생성 및 병합)
- **웹 스크래핑 최적화**: WebDriverWait, ExpectedConditions

구현 기능

- **병렬 웹 스크래핑 로직 구현**: Robots.txt 규정을 준수하며, 필터 조건에 맞는 채용 공고 정보를 병렬로 수집 후 엑셀 파일로 저장
- **페이지 분담 알고리즘 설계**: 전체 페이지를 스레드 수에 맞게 균등 분할하고, 잔여 페이지는 스레드 번호 순으로 분배하여 작업량 최적화
- **스크래핑 성능 향상**: 시스템 코어 수 기반으로 Thread Pool 크기를 동적으로 설정 → 단일 스레드 대비 약 2.1~2.4배 처리 속도 개선
- **대기 시간 최소화**: Thread.sleep() 대신 WebDriverWait + ExpectedConditions 활용으로 불필요한 대기 제거 및 페이지 반응 속도 개선
- **데이터 동기화 문제 해결**: 각 스레드가 개별 엑셀 파일을 생성하고, 모든 스레드 종료 후 병합하여 파일 동기화 (경쟁 상황)문제 해결
- **크로스 브라우저 호환성 대응**: Selenium 4.6+ 내장 WebDriver 활용해 브라우저·드라이버 버전 불일치 문제 해결
- **실시간 모니터링 기능 개발**: 사용자 모니터 해상도에 맞춰 스크래핑 상태를 실시간으로 시각화 표시

성과

- 병렬 스크래핑 구조 도입으로 단일 스레드 대비 **약 2.1~2.4배 성능 향상**
- **Excel 경쟁 조건 및 쓰레드 동기화 문제 해결** → 안정적인 데이터 병합 구현
- **페이지 로딩 속도 개선 및 스크래핑 실패율 감소** → 대규모 데이터 수집의 안정성 확보
- **실시간 진행 상황 시각화** → 사용자 피드백 향상 및 디버깅 편의성 개선

→ **총 80페이지 / 1,600건 기준, 약 2분 소요 → 40초 내외로 단축 (최대 3배 성능 향상)**

주요 키워드

robots.txt 준수 , 스크래핑 속도 병목 해결, 페이지 충돌(경쟁 상황) 방지, 엑셀 병합 자동화, 스레드 분담 최적화, 동적 요소 대기 처리, 데이터 누락/중복 방지, 수집 상태 실시간 모니터링, 외부 시스템 동기화