실무를 위한 Spring Batch의 모든 것
: 성능 튜닝부터 실전 전략까지



강사님 소개

김남윤 님

현) 국내 대기업 핀테크 BackEnd Developer
전) 스타일쉐어 BackEnd Developer
전) 소다트랜스퍼 BackEnd Developer

Blog : https://cheese10yun.github.io/
Github: https://github.com/cheese10yun



CONTENT

Part 1

Spring Batch 기초

- 스프링 배치 아키텍처
- Chunk 처리 개념
- Reader, Processor, Writer 개념

Part 2

Spring Batch 성능 최적화 기법

- Reader 성능 개선
- Processor & Writer 성능 개선
- Partition Step 성능 개선

Part 3

Spring Batch 실무에서의 배치

- 재처리가 가능한 배치 만들기
- HTTP 외부 서버 Reader
- 테스트 코드 작성 노하우

사전 지식

01 Java 언어대한 기초지식

02 Spring Framework, Spring Data JPA, JDBC의 대한 기초지식

03 Docker, MySQL의 기초지식

사용하는 기술

01 Mac OS, Jetbrains IntelliJ Ultimate

02 Java 17, Spring Boot 3, Spring Batch 5, Spring Data JPA 3, Lombok, Gradle

03 Docker Desktop, MySQL 8.0

프로젝트 코드

01 Mac OS, Jetbrains IntelliJ Ultimate

02 Java 17, Spring Boot 3, Spring Batch 5, Spring Data JPA 3, Lombok, Gradle

03 Docker Desktop, MySQL 8.0

Part 1: Spring Batch 기초

Batch 애플리케이션 특징

01 **대용량 데이터 처리:** 한 번에 대량의 데이터를 가져오거나 변환, 저장하는 처리를 수행합니다.

02 **자동화: 사용자 개입 없이** 스케줄러 등을 활용해 정해진 시간에 실행됩니다.

03 **성능:** 제한된 **시간 내 처리를 완료**하고, 동시에 실행되는 다른 애플리케이션에 지장을 주지 않도록 효율적으로 동작해야 합니다.

04 **견고성:** 잘못된 데이터나 예외 상황이 발생해도 **중단되지 않고 안정적으로 처리**할 수 있어야 합니다 (에러 시 일부 건만 건너뛰거나 재시도 등).

05 **신뢰성: 처리 과정을 추적 가능**해야 하며 오류 원인을 로깅하고 알림을 통해 확인할 수 있어야 합니다.

Batch 애플리케이션 포인트

매월 1일에

▼

이전 달 결제 금액이 1만원 이상인 회원 대상 중

▼

리뷰 이벤트를 가장 많이 작성한 회원

▼

회원 등급을 골드로 변경

Batch 애플리케이션 포인트

매월 1일에 → 스케줄링

▼

이전 달 결제 금액이 1만원 이상인 회원 대상 중

▼

리뷰 이벤트를 가장 많이 작성한 회원

▼

회원 등급을 골드로 변경

Batch 애플리케이션 포인트

매월 1일에 → 스케줄링

▼

이전 달 결제 금액이 1만원 이상인 회원 대상 중 → **Reader**

▼

리뷰 이벤트를 가장 많이 작성한 회원

▼

회원 등급을 골드로 변경

Batch 애플리케이션 포인트

매월 1일에 → 스케줄링

▼

이전 달 결제 금액이 1만원 이상인 회원 대상 중 → Reader

▼

리뷰 이벤트를 가장 많이 작성한 회원 → Processor

▼

회원 등급을 골드로 변경

Batch 애플리케이션 포인트

매월 1일에 → 스케줄링

▼

이전 달 결제 금액이 1만원 이상인 회원 대상 중 → Reader

▼

리뷰 이벤트를 가장 많이 작성한 회원 → Processor

▼

회원 등급을 골드로 변경 → Writer

Batch 애플리케이션 종류

Batch Insert

- Reader : 데이터 읽기
- Processor: 가공 처리
- Writer: 데이터 Insert

Batch Update

- Reader : 데이터 읽기
- Processor: 가공 처리
- Writer: 데이터 Update

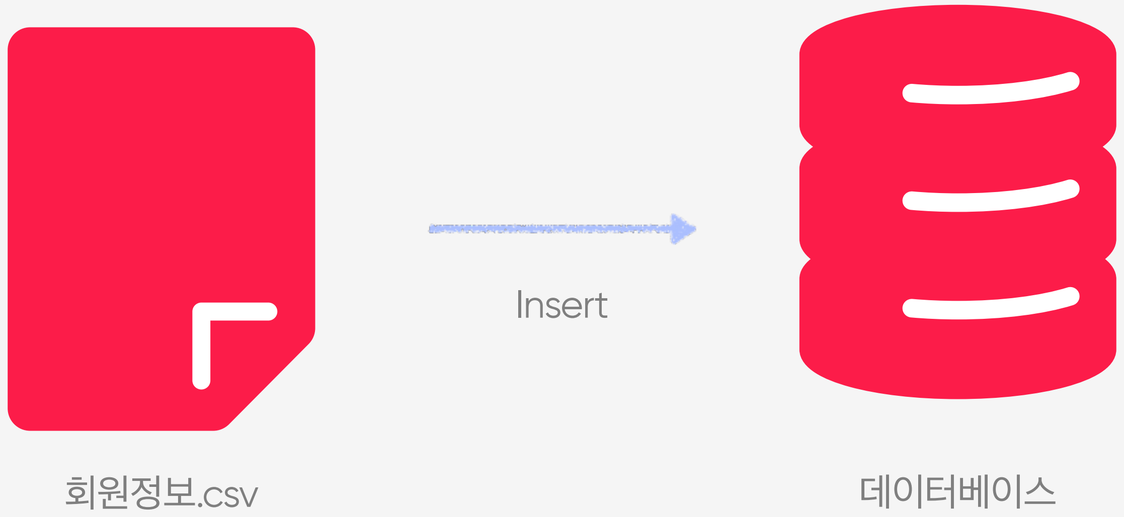
Batch Processor

- Reader : 데이터 읽기
- Processor: 가공 처리
- Writer: 가공 처리 전달

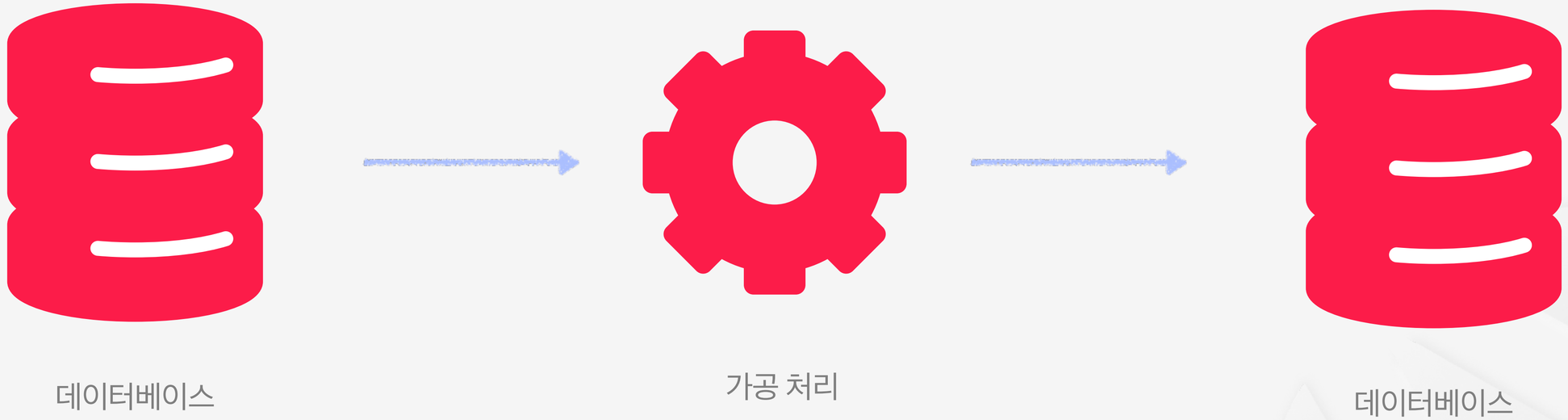
Batch Aggregation

- Reader : 데이터 읽기
- Processor: 가공 처리
- Writer: 집계 처리(Sum, Group By)

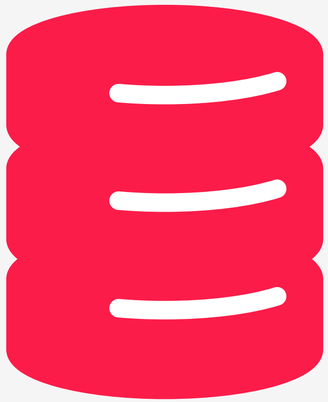
Batch 애플리케이션 종류 - Batch Insert



Batch 애플리케이션 종류 - Batch Update

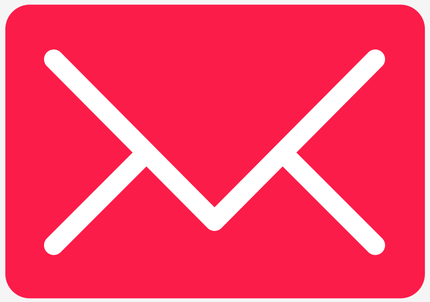


Batch 애플리케이션 종류 - Batch Processor



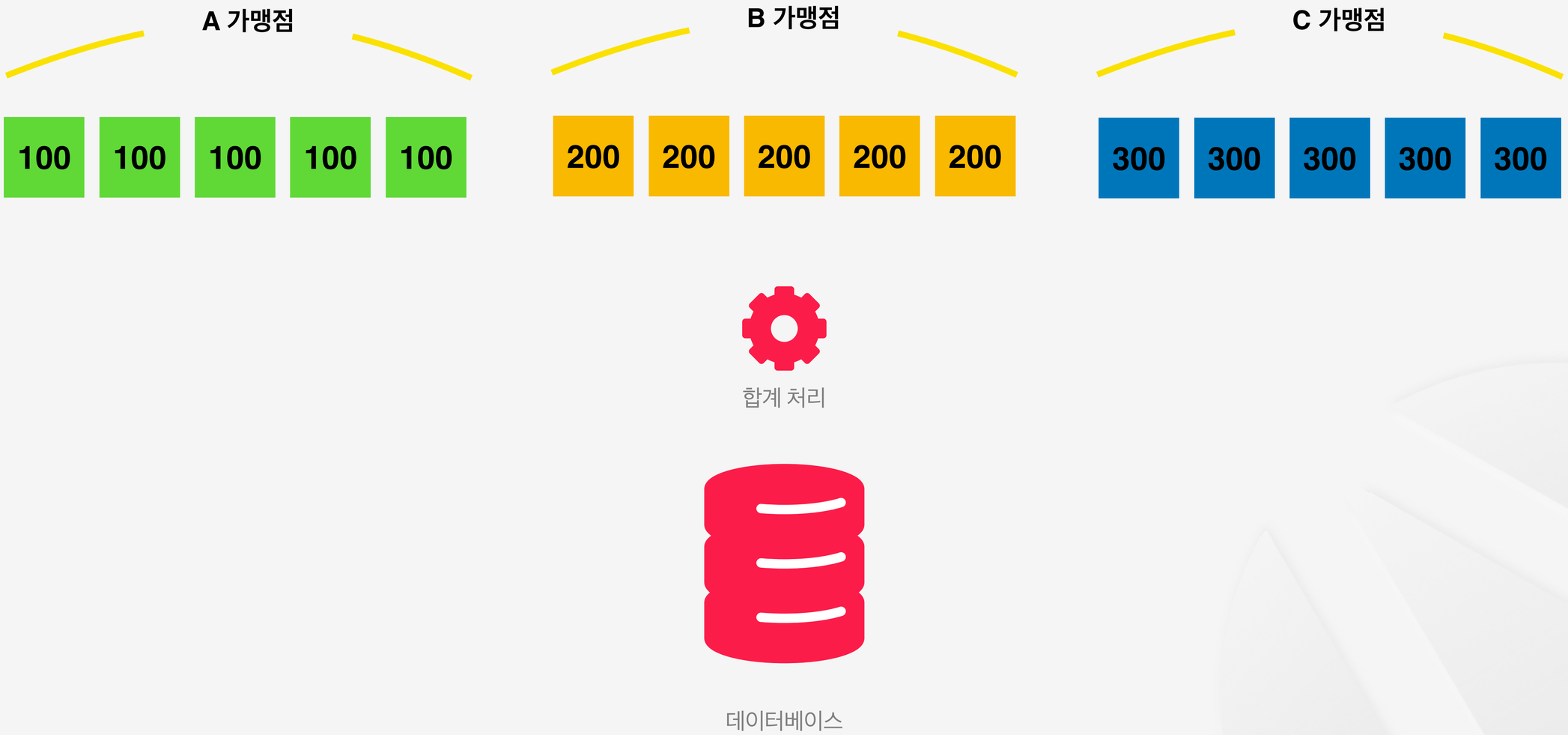
데이터베이스





이메일 전송

Batch 애플리케이션 종류 - Batch Aggregation



Batch 애플리케이션 종류

수정

단순 수정 배치: 기존 데이터의 특정 필드를 일괄 변경하거나 상태 플래그를 업데이트하는 작업입니다.
예를 들어, **만료된 쿠폰의 상태를 일괄적으로 '만료됨'으로 업데이트**하거나, **장기간 로그인하지 않은 계정의 활성화 여부를 N으로 변경**하는 등의 작업이 단순 수정 배치에 해당합니다.

저장

외부 또는 신규 데이터를 가져와 저장하는 작업입니다.
예로 **대용량 파일(CSV/XML) 데이터를 데이터베이스에 적재**하는 배치, **다른 시스템의 데이터를 주기적으로 수집하여 저장**하는 배치를 들 수 있습니다.
예를 들어 **"매일 서버 로그 파일을 읽어 DB의 로그 테이블에 적재"**, **"API를 호출해 받아온 데이터를 DB에 누적 저장"** 등이 있습니다.

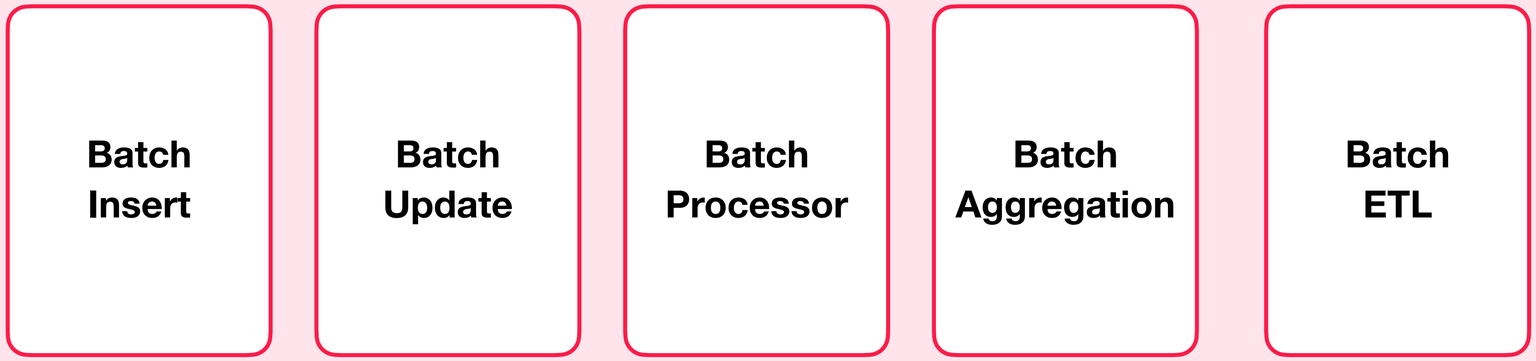
집계

집계(aggregation) 배치: 다수의 데이터를 읽어 요약 정보나 통계를 산출하는 배치입니다.
예를 들어 "일별 판매 데이터를 모아 총 매출과 거래 건수를 계산하여 리포트 생성", "사내 시스템의 로그를 분석해 오류 발생 빈도 집계" 등이 있습니다.

ETL

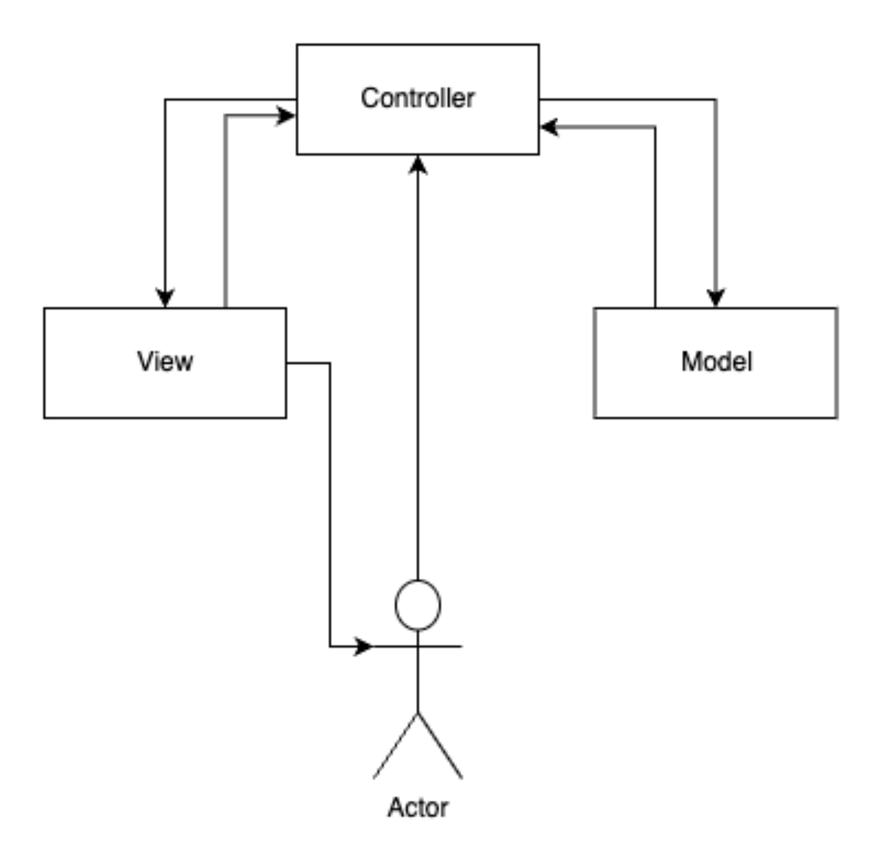
데이터 웨어하우스나 마이그레이션에서 흔한 유형으로, **추출-변환-적재** 과정을 거치는 배치입니다.
한 시스템에서 데이터를 추출(Extract)하여, 필요한 형태로 가공(Transform)하고, 다른 시스템/파일로 적재(Load)하는 전형적인 업무입니다.
예를 들어 **"레거시 DB에서 데이터를 추출해 새로운 시스템 DB로 변환/이관"**, **"여러 소스의 데이터를 추출해 정규화된 형태로 변환 후 통합 저장"** 등이 있습니다.

Batch 애플리케이션 종류 - 조합

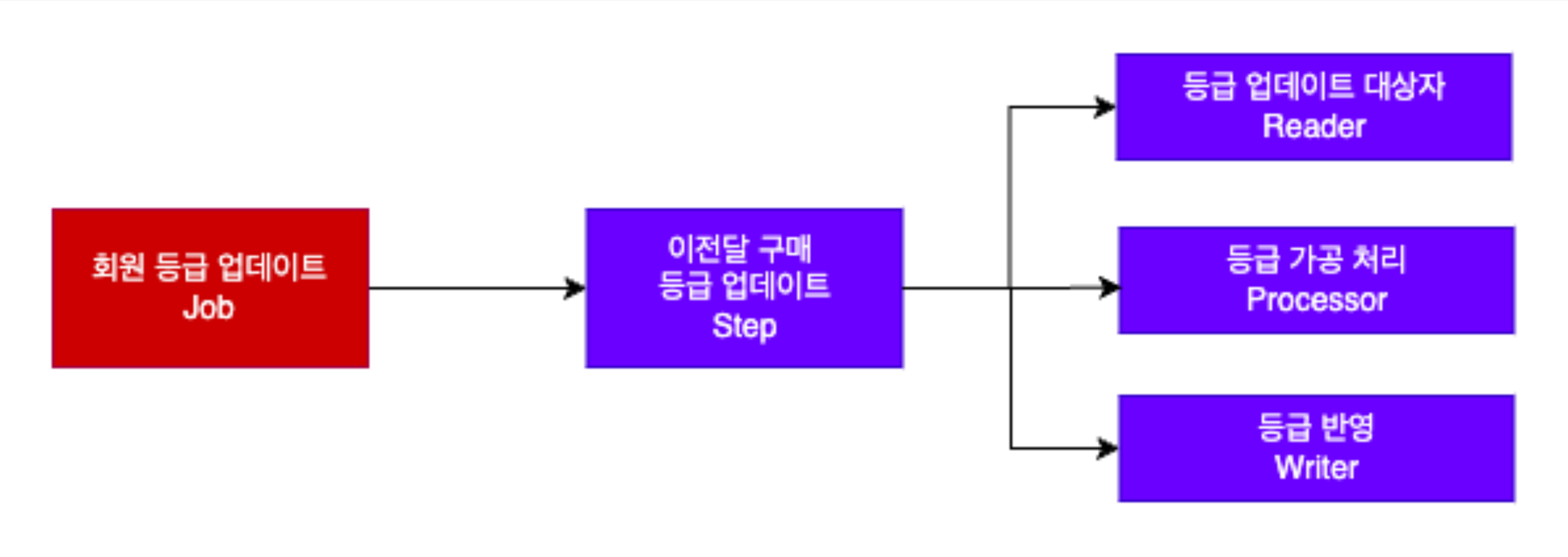


왜 Spring Batch 인가?

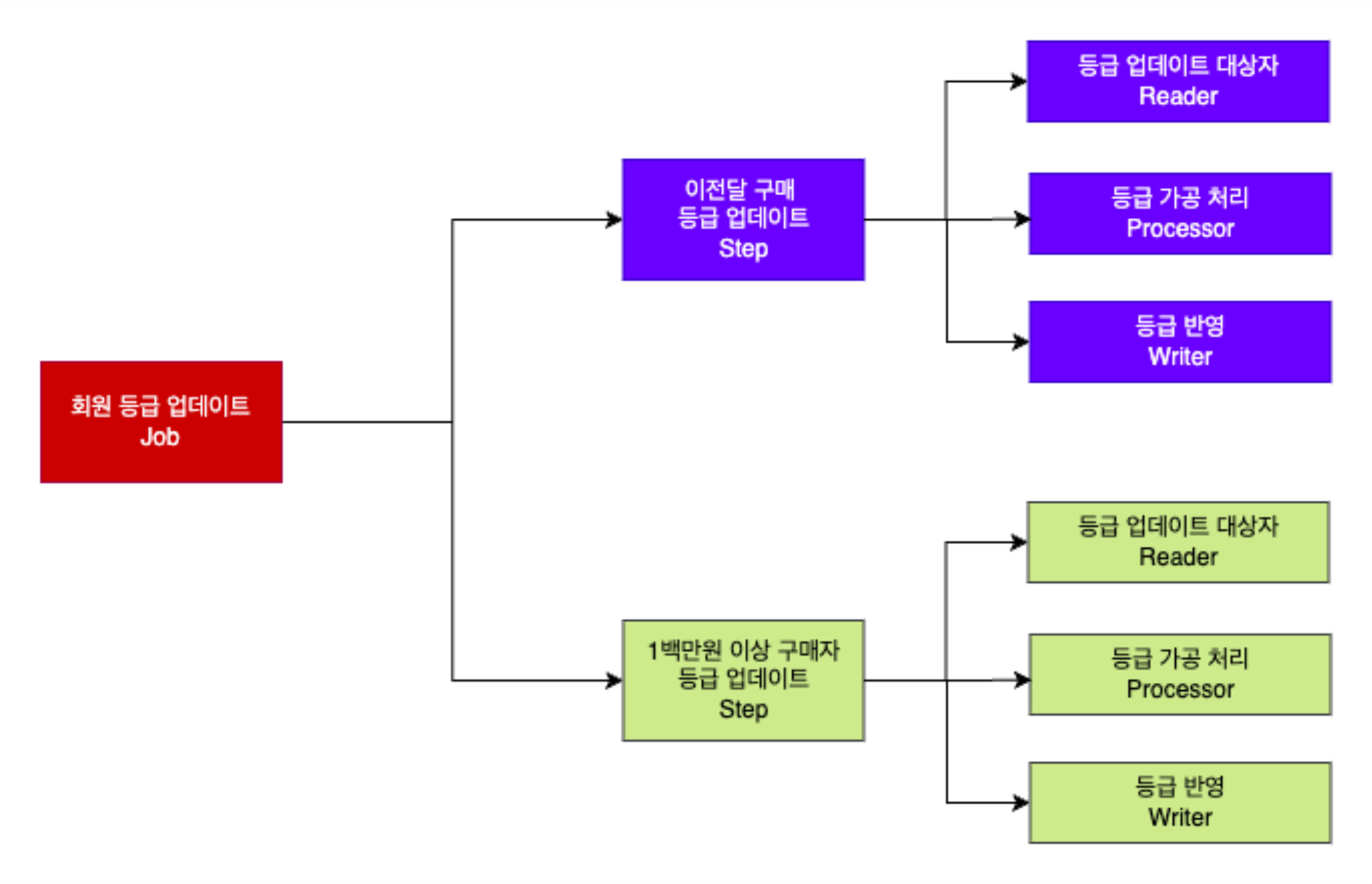
Model View Controller - MVC Pattern



Spring Batch Structure



Spring Batch Structure



Spring Batch 특징 및 장점

1. 풍부한 필수 기능 제공

- 트랜잭션 관리, 로깅/추적, 체크포인트/재시작, 통계 집계, 예외 처리(Skip), 재시도(Retry) 등의 기능을 간편히 활용 가능
- 실패한 배치 Job을 중단 지점부터 재시작하거나 동일 파라미터로 중복 실행 방지 지원

2. Spring 생태계와 쉬운 통합

- Spring Framework 기반으로 DI, AOP 활용 가능
- Spring Boot 환경에서 Starter 의존성 추가만으로 자동 설정(DataSource, JobRepository 등) 제공
- Spring Integration, Spring Cloud(Data Flow, Task)와 연계하여 배치 파이프라인 확장 및 모니터링 가능

3. 검증된 성능과 안정성

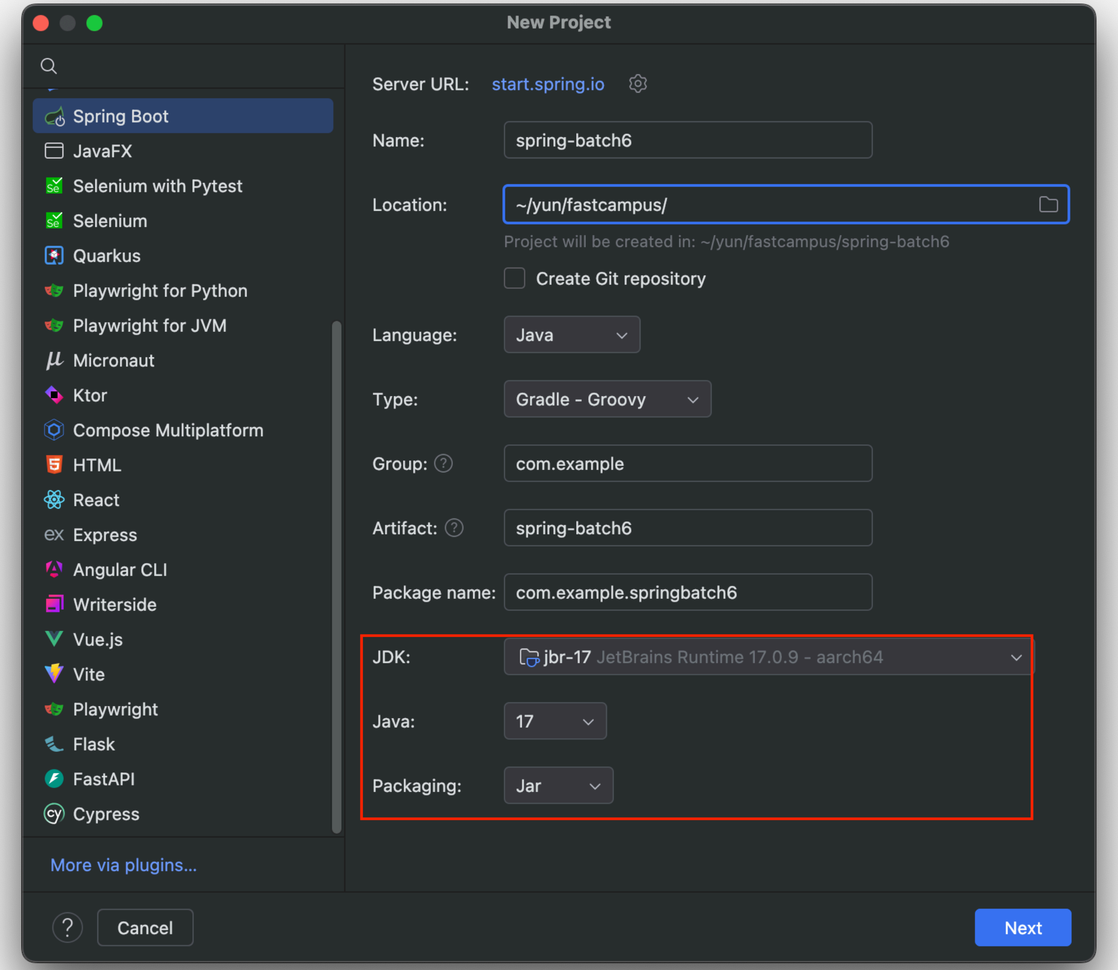
- 금융권, 이커머스 등 다양한 산업에서 검증된 신뢰성과 성능 제공
- 청크 기반 처리, 멀티스레딩, 파티셔닝을 통한 효율적인 대용량 데이터 처리 지원
- 청크 크기, 스레드 개수 등 세부적인 성능 튜닝 가능

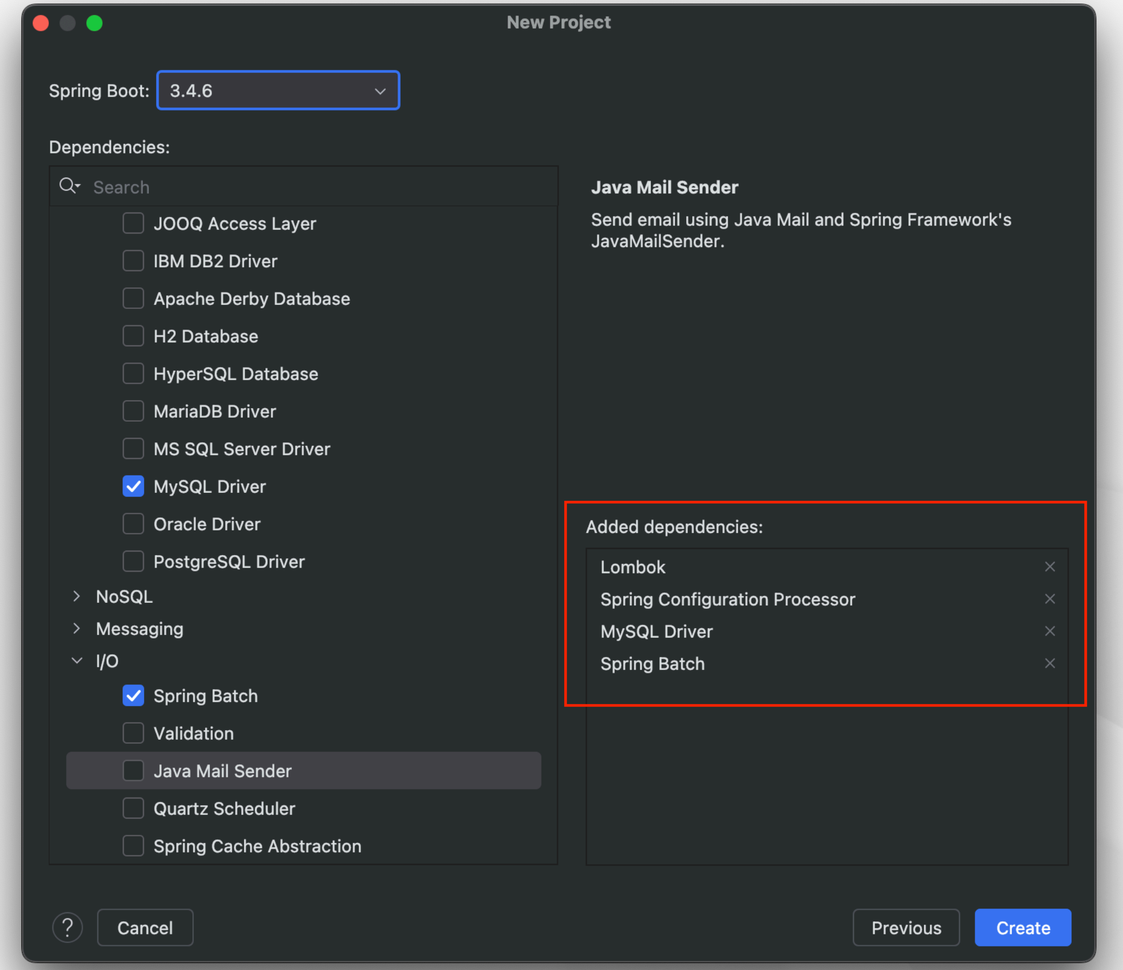
4. 다양한 **Reader/Writer** 지원

- 파일(CSV, XML, JSON), DB(JDBC, JPA), JMS, MongoDB 등의 다양한 데이터 소스를 기본적으로 지원
- 커스텀 ItemReader/Writer로 새로운 데이터 소스에 쉽게 대응 가능

개발 환경 셋업 및 첫 배치 Job 실행

프로젝트 설정


New Project
Spring Boot
JavaFX
Selenium with Pytest
Selenium
Quarkus
Playwright for Python
Playwright for JVM
Micronaut
Ktor
Compose Multiplatform
HTML
React
Express
Angular CLI
Writerside
Vue.js
Vite
Playwright
Flask
FastAPI
Cypress
More via plugins...
Server URL: start.spring.io
Name: spring-batch6
Location: ~/yun/fastcampus/
Project will be created in: ~/yun/fastcampus/spring-batch6
Create Git repository
Language: Java
Type: Gradle - Groovy
Group: com.example
Artifact: spring-batch6
Package name: com.example.springbatch6
JDK: jbr-17 JetBrains Runtime 17.0.9 - aarch64
Java: 17
Packaging: Jar
Cancel
Next


New Project
Spring Boot: 3.4.6
Dependencies:
Search
JOOQ Access Layer
IBM DB2 Driver
Apache Derby Database
H2 Database
HyperSQL Database
MariaDB Driver
MS SQL Server Driver
MySQL Driver
Oracle Driver
PostgreSQL Driver
NoSQL
Messaging
I/O
Spring Batch
Validation
Java Mail Sender
Quartz Scheduler
Spring Cache Abstraction
Java Mail Sender
Send email using Java Mail and Spring Framework's JavaMailSender.
Added dependencies:
Lombok
Spring Configuration Processor
MySQL Driver
Spring Batch
Cancel
Previous
Create

프로젝트 설정 - docker mysql

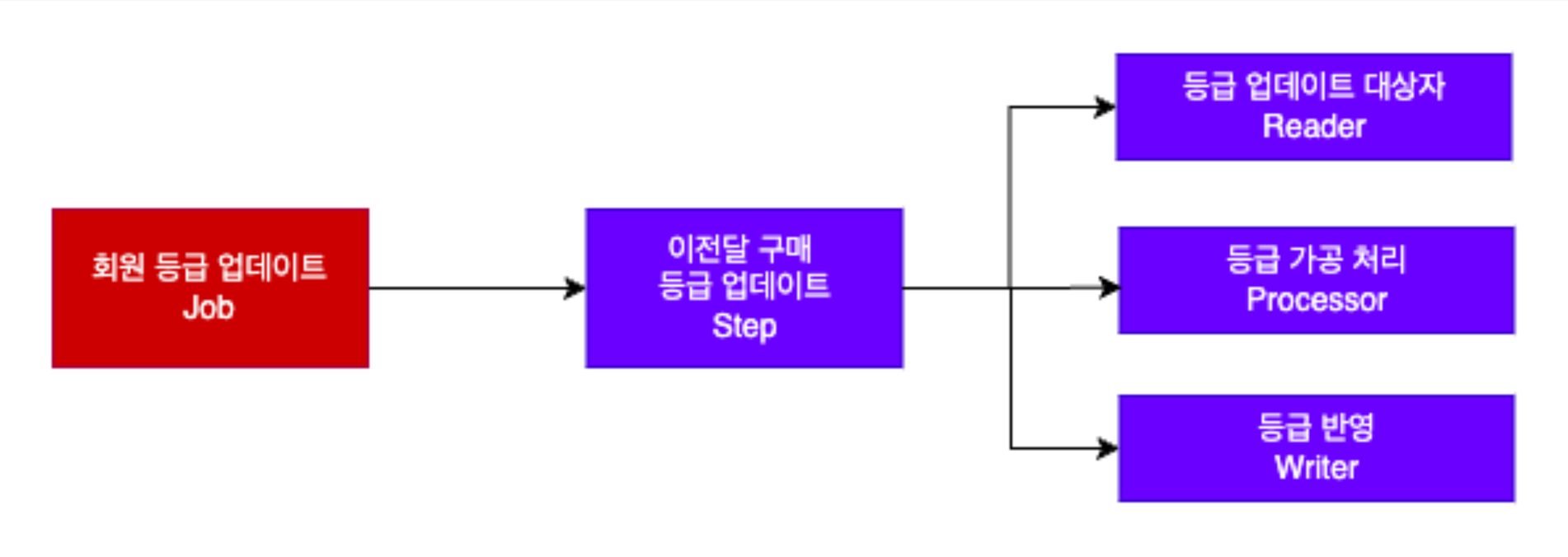
compose.yaml

```yaml
services:
    mysql:
        container_name: 'mysql-spring-batch'
        image: 'mysql:8.0.34'
        environment:
            MYSQL_ROOT_HOST: '%'
            MYSQL_DATABASE: 'spring-batch'
            MYSQL_ALLOW_EMPTY_PASSWORD: 'yes'
            TZ: 'Asia/Seoul'
        ports:
            - '3306:3306'
        volumes:
            - './volumes/mysql/default:/var/lib/mysql'
        command:
            - "--character-set-server=utf8mb4"
            - "--collation-server=utf8mb4_unicode_ci"
```

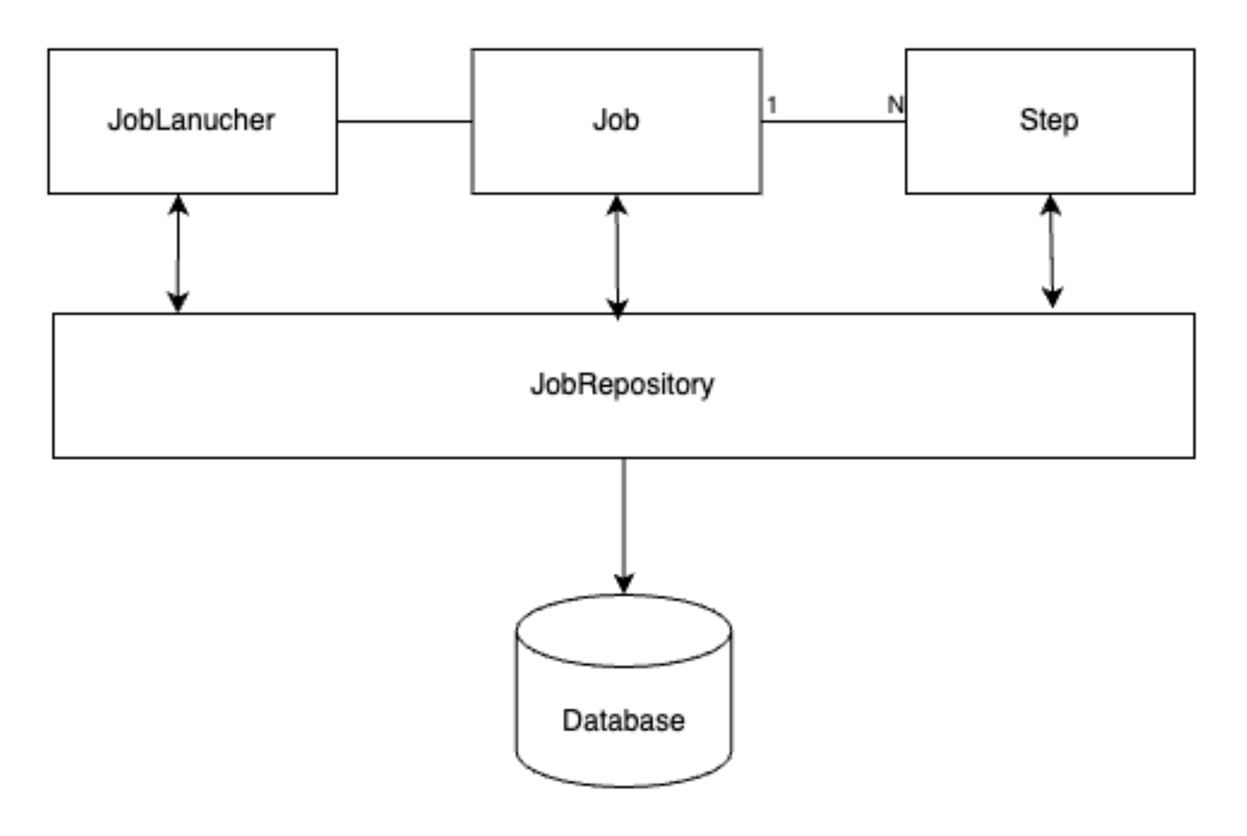
- `container_name`: 컨테이너 이름 설정 ('mysql-spring-batch')
- `image`: 사용할 MySQL 이미지 버전 지정 (8.0.34)
- `environment`: 컨테이너 내부 환경 변수 설정
  - `MYSQL_ROOT_HOST`: root 계정의 접속 허용 범위 (모든 호스트 허용)
  - `MYSQL_DATABASE`: 시작 시 생성될 데이터베이스 이름
  - `MYSQL_ALLOW_EMPTY_PASSWORD`: 빈 비밀번호 허용 여부 (개발 환경용)
  - `TZ`: 컨테이너의 타임존 설정
- `ports`: 포트 매핑 (호스트의 3306 포트를 컨테이너의 3306 포트에 연결)
- `volumes`: 데이터 지속성을 위한 볼륨 설정 (로컬 경로를 MySQL 데이터 디렉토리에 마운트)
- `command`: 컨테이너 시작 시 실행할 추가 명령어 (문자셋과 정렬 규칙 설정)

Spring Batch 구조

Spring Batch 구조



Spring Batch 구조



JobLauncher - 배치 잡(Job)을 실행하는 진입점(Entry Point)

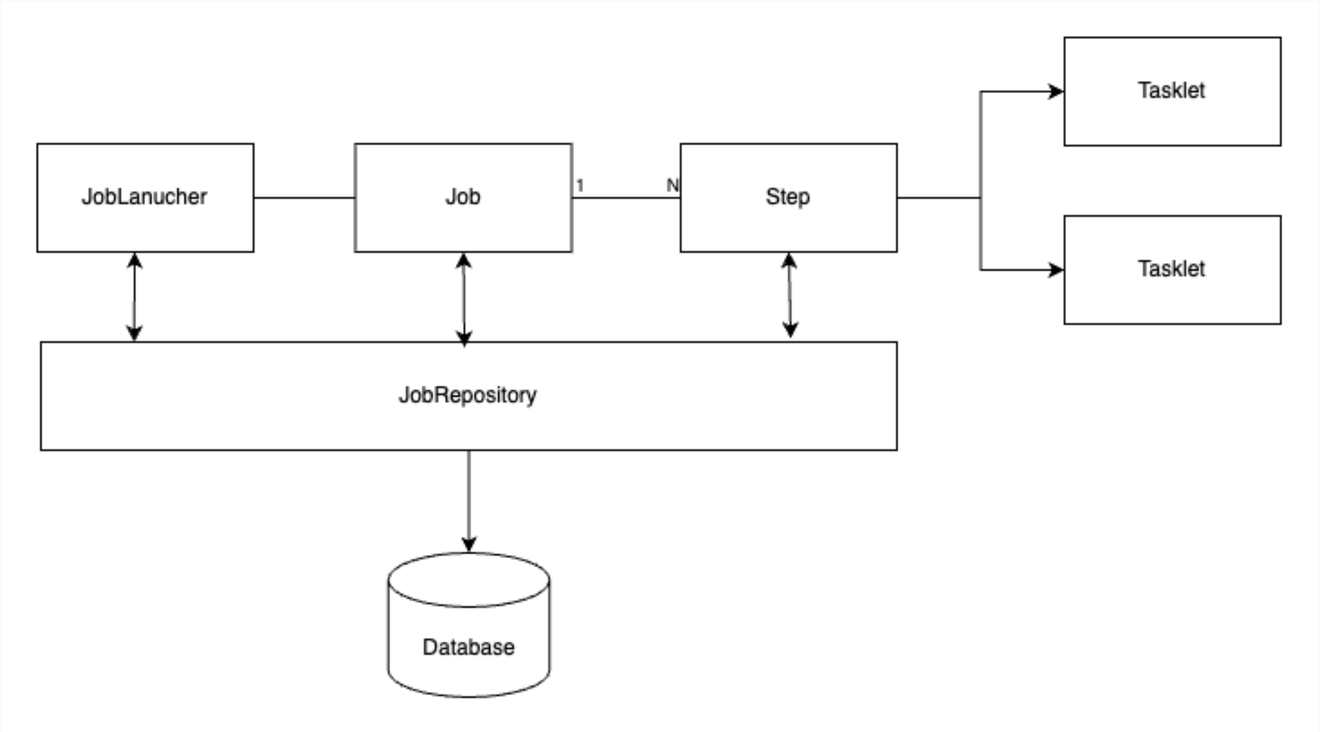
Job - 하나의 배치 작업 단위, 내부에 여러 개의 Step의로 구성

Step - Chunk 기반 처리나 Tasklet 단위로 비즈니스 로직을 수행

JobRepository - 배치 메타데이터(실행 기록, 파라미터, 상태 등)를 저장·조회하는 저장소

Database - 배치 메타데이터 테이블이 생성·관리되는 물리적 저장소

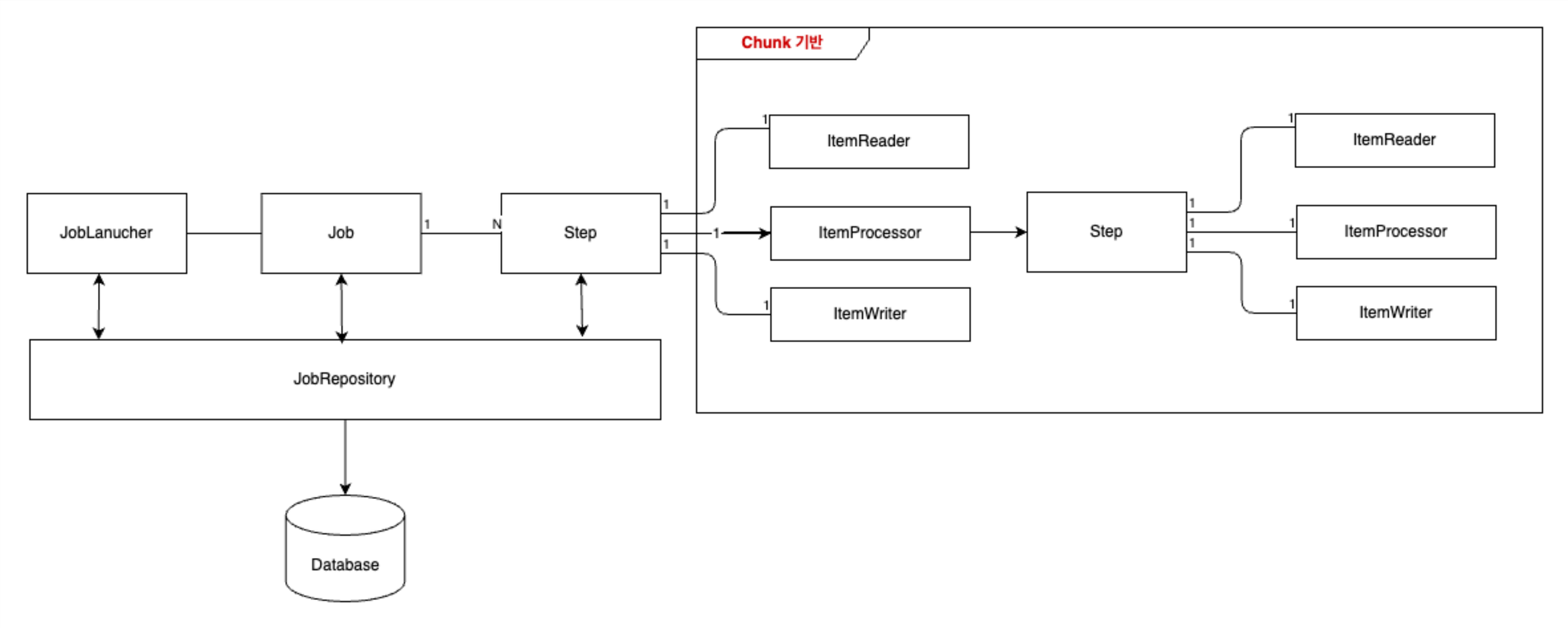
Spring Batch 구조 - Tasklet



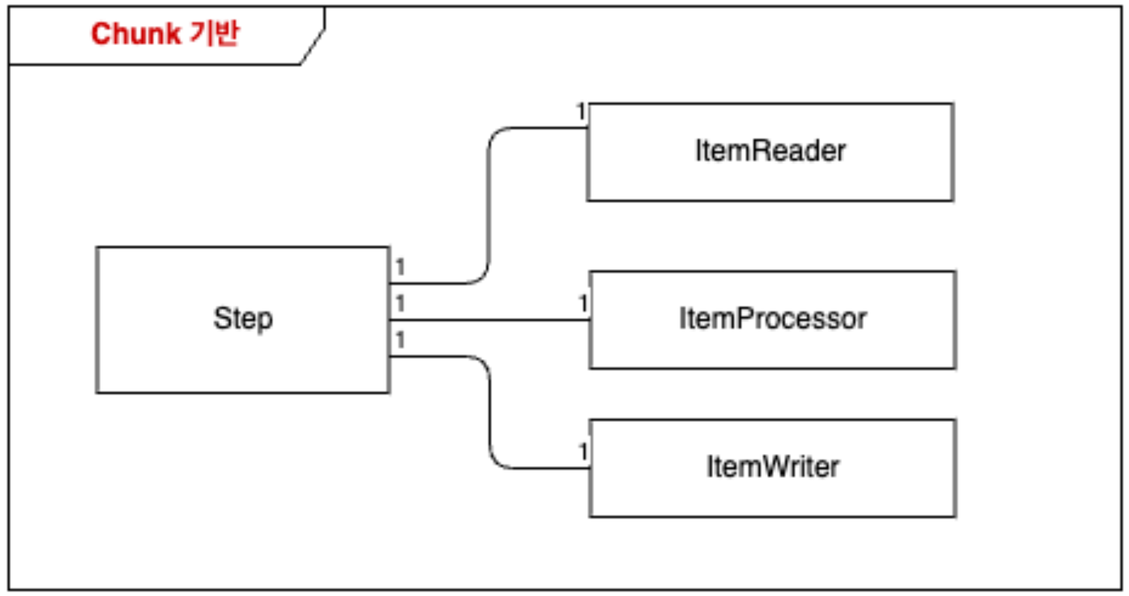
Tasklet - Step 내에서 단일 작업 단위를 수행하는 컴포넌트

Tasklet - 적용 예시: 외부 API 호출, 시작/종료 알림 전송 등

Spring Batch 구조 - Chunk



Spring Batch 구조



ItemReader - 처리 대상을 읽어오는 단계

ItemProcessor - 읽어온 아이템을 가공·변환·검증하는 단계

ItemWriter - 가공된 아이템을 **저장**하거나 전송

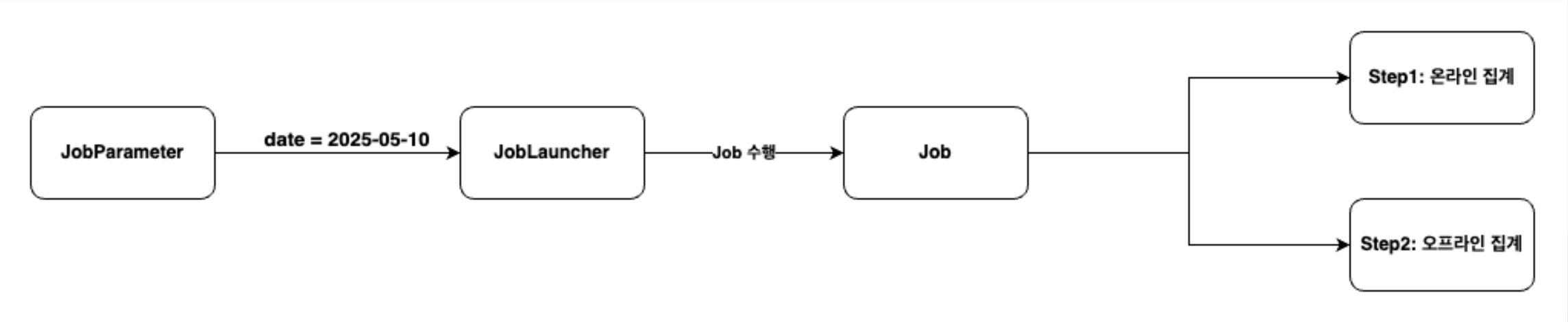
Chunk 기반 - 데이터를 처리하는 묶음의 크기

Job 및 Step 심화 이해

Job -기본개념

- Job은 Spring Batch에서 배치 처리의 최상위 단위입니다.
  - "어제 발생한 주문 데이터를 집계해서 리포트로 저장한다”
  - "정산 대상 데이터를 추출하고 정산을 진행한다”
  - "외부 API로부터 데이터를 수집한다”
  - 즉, 하나의 Job = 배치 처리 단위 하나, 내부에 여러 개의 Step이 포함될 수 있고, Job은 Step들을 조합하여 전체 배치 흐름을 정의합니다.
- Spring Batch에서 Job은 구성(configuration)과 실행(execution) 두 역할을 동시에 수행합니다.
  - Job은 어떤 Step들을 어떤 순서로 실행할지 정의합니다.
  - Job은 JobLauncher에 의해 실행되며, 실행 시 전달된 JobParameters를 기반으로 고유한 JobInstance를 생성합니다.
  - 이 실행은 JobExecution이라는 실행 이력으로 기록되며, 추적 및 재시작이 가능합니다.

Job -기본개념


JobParameter
date = 2025-05-10
JobLauncher
Job 수행
Job
Step1: 온라인 집계
Step2: 오프라인 집계

Job

- Job 이란?
  - **Job**은 배치 처리의 “설계도” 혹은 “전체 플로우의 최상위 단위”입니다.
  - 한 번의 Job 실행 = “비즈니스 목적을 달성하는 하나의 완결된 배치 작업”
- 구성 예시
  - Step1: 전날 주문 데이터 읽기 → 집계
  - Step2: 집계 결과를 파일로 저장
  - Step3: 파일을 외부 서버로 전송
- 특징
  - Job마다 반드시 이름이 필요함 → 메타데이터 테이블의 PK, 중복 실행 방지, 추적의 기준
  - Job은 재시작/중단/상태 관리가 가능(실패 시 중단된 위치부터 재실행 가능)
  - 실행 이력, 성공/실패 상태, 종료 코드 등이 메타 테이블에 자동 기록됨

JobInstance

- JobInstance 이란?
  - JobInstance = “특정 Job이 특정 파라미터로 실행된 논리적 인스턴스”
  - 같은 Job이라도, 실행 파라미터(예: date)가 다르면 새로운 JobInstance로 간주
  - 즉, Job명 + JobParameters 조합으로 유일함
- 특징
  - 같은 JobName, 같은 JobParameters면 이미 존재하는 JobInstance → **중복 실행 방지**
  - 다른 JobParameters면, 매번 새로운 JobInstance 생성
  - JobInstance가 생성되면, 여러 번의 실행(JobExecution)과 연결될 수 있음(실패/재시작 시)
- 실무에서 유용한 점
  - JobInstance 기준으로 실행 이력을 관리하므로, 예를 들어 “2024-06-01자 집계 작업을 실패 후 재시작할 때” 같은 논리적 묶음 관리 가능
  - 메타 테이블(JOB_INSTANCE 테이블 등)에서 상태/실행 이력 추적 가능

JobParameters

- JobParameters 이란?
  - Job 실행 시 입력되는 파라미터 집합
  - JobInstance의 유일성을 결정짓는 핵심 요소
  - 실행 시점, 대상 일자, 파일명 등 작업을 구분하는 데 쓰임
- JobParameters와 JobInstance의 관계
  - JobName + JobParameters의 조합이 기존에 존재하면, 동일 JobInstance로 인식
  - 파라미터 하나라도 다르면 새로운 JobInstance 생성 (예: date 값만 달라져도 새로운 작업으로 취급)
- 메타 테이블 반영
  - JOB_INSTANCE 테이블의 JOB_KEY 컬럼에 해시값 등으로 기록
  - 모든 Job 실행 이력의 출발점이 되는 값
  - 실패/재시작/중복 방지 등 모든 상태관리의 기준

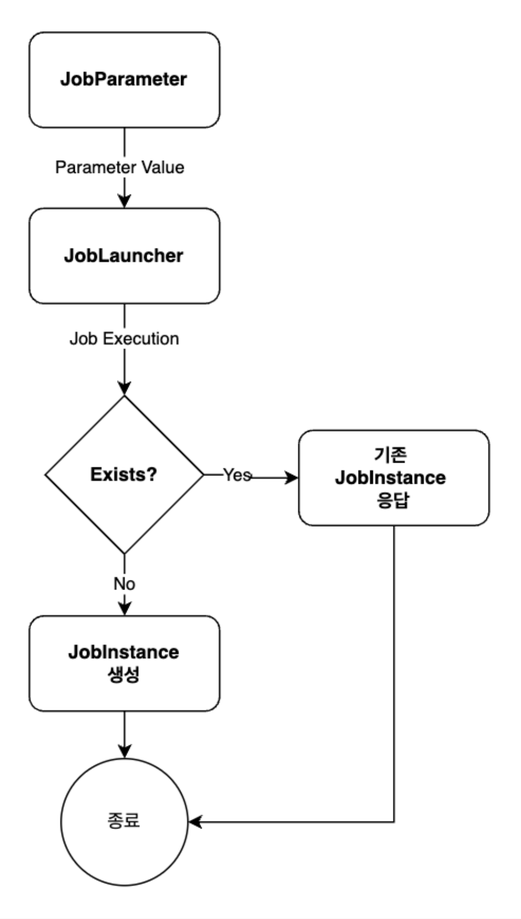
JobExecution

- JobExecution 이란?
  - JobExecution = JobInstance의 한 번의 실제 실행 이력(Record)
  - 하나의 JobInstance(특정 JobName+JobParameters)에 대해 여러 번 실행(재시작, 실패 후 재시도 등)이 있을 수 있음
  - 실행마다 별도의 JobExecution이 생성되고, 각각 status(성공/실패), 시작/종료 시각, 에러메시지, 종료코드 등이 저장
- 주요 특징
  - JobExecution이 쌓이면 “언제, 누가, 어떤 조건으로 실행했고, 성공/실패했는지” 전부 기록
  - 배치 실패 원인 추적, SLA(처리시간 보장), 장애 대응, 이중실행 방지 등에서 매우 중요
- 메타 테이블 반영
  - JOB_INSTANCE 테이블의 JOB_KEY 컬럼에 해시값 등으로 기록
  - 모든 Job 실행 이력의 출발점이 되는 값
  - 실패/재시작/중복 방지 등 모든 상태관리의 기준

Job 구현체

- Spring Batch – Job의 기본 구현체
  - Spring Batch는 Job 인터페이스의 대표적인 두 가지 구현체를 제공합니다.
  - 각각의 Job 구현체는 Step을 어떤 방식으로 실행할지, 그리고 흐름을 어떻게 제어할지에 따라 구분됩니다.
- SimpleJob: 가장 기본적인 Job 구현체
  - 내부에 정의된 Step들을 **순차적**으로 한 줄 흐름으로 실행합니다.
  - 추가적인 조건 분기 없이 **선형적인 작업 흐름**을 처리할 때 적합합니다.
- FlowJob: 복잡한 흐름 제어(조건 분기, 병렬 흐름 등)가 필요한 경우 사용되는 구현체입니다.
  - Flow 객체를 기반으로 실행 흐름을 구성하며, StepA 성공 시 StepB, 실패 시 StepC 실행 등
  - **조건에 따라 분기하거나,** 특정 Step을 재시도, 병렬 실행, skip하는 등 보다 **정교한 시나리오 설계**에 적합합니다.

Job 실행 및 메타데이터 관리



BATCH_JOB_INSTANCE

JOB_INSTANCE_ID	JOB_NAME	JOB_KEY
2	paymentReportJob	d41d...
3	paymentReportJob	9a6...

BATCH_JOB_EXECUTION_PARAMS

JOB_EXECUTION_ID	PARAMETER_NAME	PARAMETER_VALUE
2	targetDate	2025-05-11
3	targetDate	2025-05-11
3	run.id	1

Step

- Step이란?
  - Step은 Spring Batch에서 '실제 데이터 처리'가 일어나는 최소 단위입니다.
  - 하나의 Job은 여러 개의 Step으로 이루어질 수 있고, 각 Step이 순차적 혹은 조건에 따라 실행되면서 전체 배치 작업이 완성됩니다.
  - Step 내부에서는 Reader → Processor → Writer의 구조로 데이터를 읽고(Reader), 가공하고(Processor), 저장(Writer)합니다.
  - Step이 실행될 때마다 처리 건수, 성공/실패, 상태 등이 메타테이블에 기록되어 모니터링, 장애 대응, 재시작에 활용됩니다.
  - Step은 Tasklet 기반(단일 작업) 또는 Chunk 기반(데이터 묶음 단위 반복 처리)으로 구현할 수 있습니다.

Step

- TaskletStep
  - 하나의 Step에서 '특정 로직'을 단일 작업으로 실행할 때 사용
  - ex) 파일 삭제, 외부 시스템 알림 전송, 단건 처리 등
  - Tasklet 인터페이스를 구현해서 원하는 로직만 작성하면 됨
- ChunkOrientedTasklet (Chunk 기반 Step, 실무에서 가장 많이 사용)
  - 대용량 데이터를 'Chunk'(묶음) 단위로 반복 처리 예) 1000건 데이터를 100건씩 나누어 읽고, 가공, 저장 반복
  - 내부적으로 ItemReader → ItemProcessor → ItemWriter 구조를 따름
  - 데이터 집계, 이관, 대량 파일 처리 등 대부분의 배치 처리에 사용

StepExecution

- StepExecution 이란?
  - Step의 한 번의 실행(Record), 즉 실행 이력 단위.
  - 언제, 어떤 상태로 실행됐고, 성공/실패/중단 등 결과가 어떻게 됐는지 모두 기록
- StepExecution의 주요 속성 및 활용성
  - 실행 시작/종료 시간, 상태(성공, 실패 등), 처리/읽은/쓰여진 건수, 에러 메시지 등
  - 각 StepExecution별 처리 건수 모니터링
  - 장애 발생 시, 어느 Step에서 문제가 발생했는지 추적
- 메타 테이블과의 연관
  - BATCH_STEP_EXECUTION 테이블에 이력 저장

StepContribution

- StepContribution 이란?
  - StepExecution 내부에서 한 Chunk(묶음)의 처리 결과를 누적·집계하는 역할.
  - 청크 기반 처리에서 각 Chunk별로 Reader/Processor/Writer의 처리 결과를 기록하고, StepExecution에 반영
- StepContribution의 주요 속성
  - 읽은/처리한/쓰여진/스킵된 아이템 수
  - 예외 발생, 필터링, 실패 건수 등 상세 통계
  - 성능 튜닝, 장애 분석, 청크 사이즈 조정 시 StepContribution 로그가 매우 유용함
- 실제 동작 흐름
  - 청크 사이즈마다 ItemReader → ItemProcessor → ItemWriter가 실행되고, 그 결과가 StepContribution에 저장됨
  - 모든 Chunk가 끝나면 StepContribution의 통계 정보가 StepExecution에 합산됨

Schema BATCH_STEP_EXECUTION

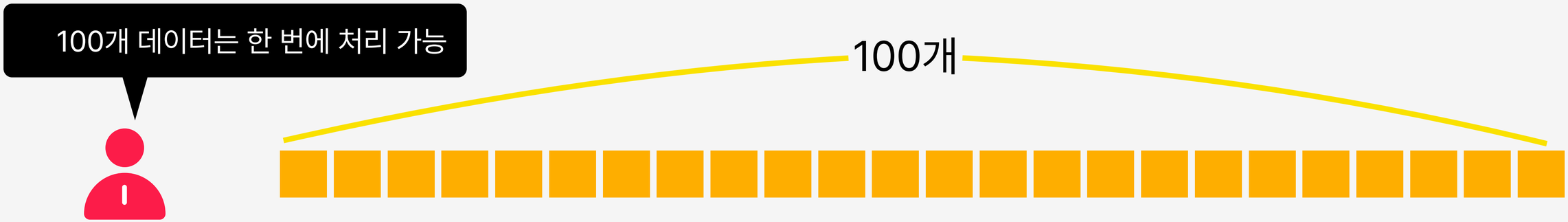
STEP_EXECUTION ID	STEP_NAME	START_TIME	END_TIME	STATUS	COMMIT COUNT	READ COUNT	FILTER COUNT	WRITE COUNT	READ_SKIP COUNT	WRITE_SKIP COUNT	PROCESS_SKIP COUNT	ROLLBACK COUNT	EXIT_CODE
1	paymentReportStep	2025-05-30 ...	2025-05-30 ...	COMPLETED	3	14	0	14	0	0	0	0	COMPLETED
2	paymentReportStep	2025-05-30 ...	2025-05-30 ...	COMPLETED	3	14	0	14	0	0	0	0	COMPLETED
3	paymentReportStep	2025-05-30 ...	2025-05-30 ...	COMPLETED	3	14	0	14	0	0	0	0	COMPLETED
4	paymentReportStep	2025-05-30 ...	2025-05-30 ...	COMPLETED	3	14	0	14	0	0	0	0	COMPLETED

Chunk 기반 처리

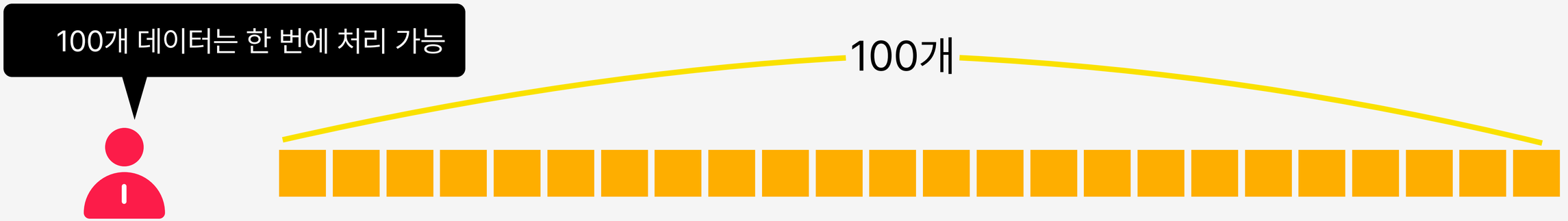
Chunk 기반 처리

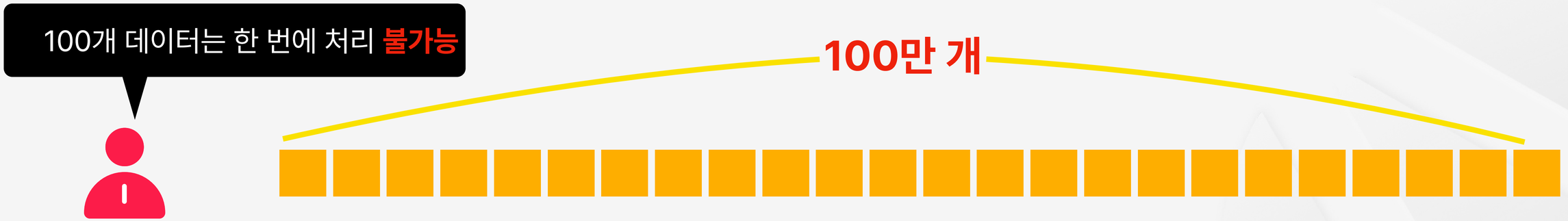
- Chunk란?
  - 대용량 데이터를 '덩어리(Chunk, 묶음)' 단위로 잘라서 일정 개수씩 반복 처리하는 방식입니다.
  - 예를 들어, 1000건의 데이터를 100건씩 10번에 나누어 처리(읽기→가공→쓰기)를 반복합니다.
- 처리 구조
  - Step 내부에서 ItemReader → ItemProcessor → ItemWriter 구조로 동작합니다.
  - Reader에서 데이터를 Chunk 사이즈만큼 읽고, Processor에서 가공/필터링한 뒤, Writer에서 일괄 저장합니다.
- 장점
  - 메모리 효율성: 한 번에 모든 데이터를 올리지 않고, Chunk 단위로 나눠서 처리하므로 메모리 부담이 적습니다.
  - 성능 향상: DB/파일 등에 대량의 데이터를 일괄 저장/커밋하여 I/O 성능을 극대화할 수 있습니다.
  - 트랜잭션 관리: 각 Chunk 단위로 트랜잭션이 걸리므로 실패 시 해당 Chunk만 재시도/롤백할 수 있습니다.
  - 대부분의 배치 처리에 표준적으로 사용: 데이터 집계, 이관, 파일 처리 등 다양한 실무에서 널리 쓰입니다.
- 예시
  - Step1: CSV 파일을 100건씩 읽어서 DB에 저장
  - Step2: DB 데이터를 200건씩 집계하여 리포트 파일 생성

Chunk 단위 분할 처리 방식 이해하기

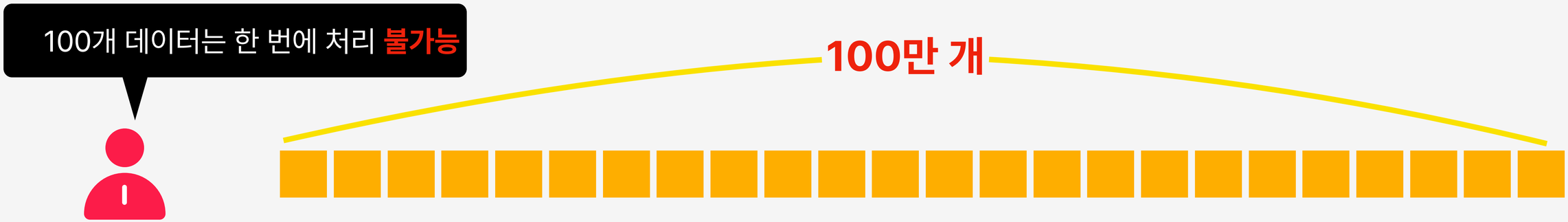


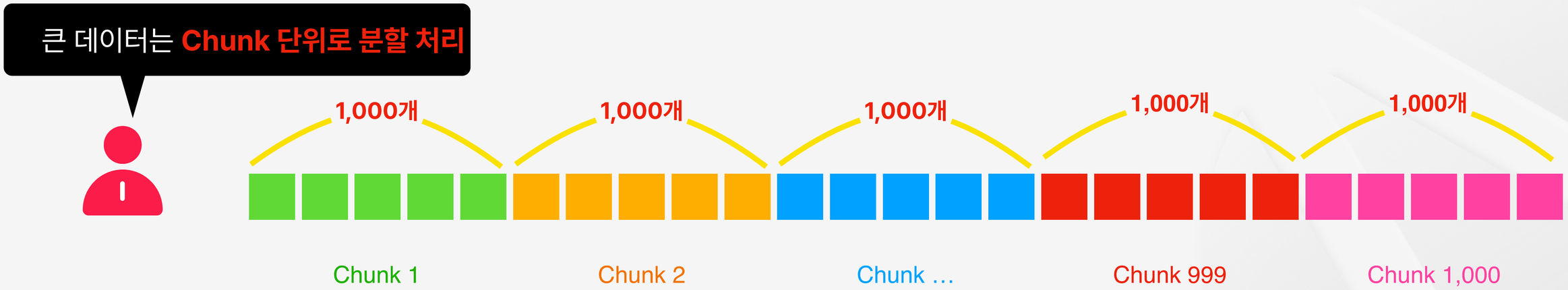
Chunk 단위 분할 처리 방식 이해하기



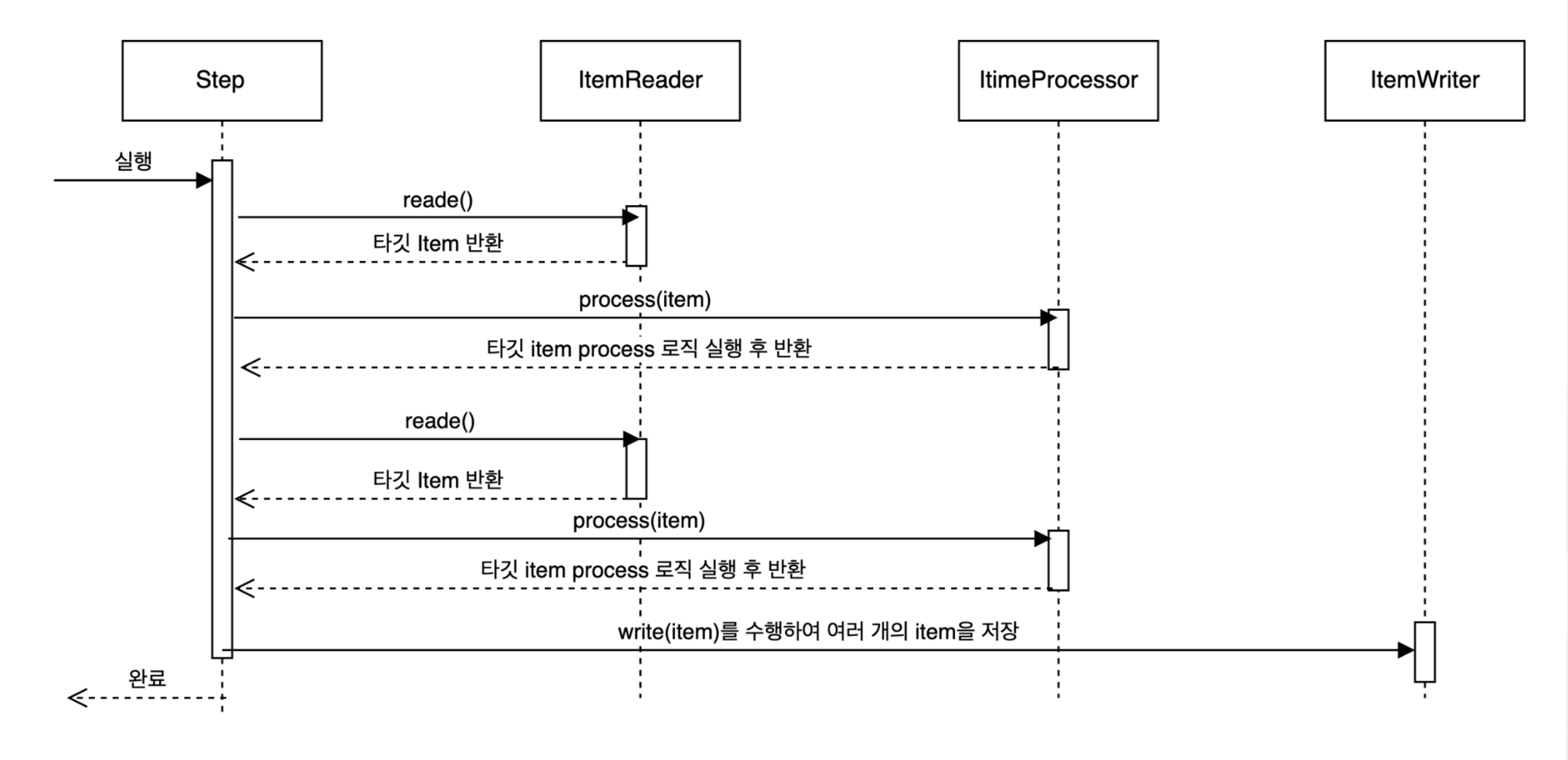


Chunk 단위 분할 처리 방식 이해하기





Chunk 프로세싱 시퀀스 흐름



Chunk 프로세싱 코드 흐름

```
1 public void Chunk_처리_방법(int chunkSize, int totalSize) {
2     for (int i = 0; i < totalSize; i = i + chunkSize) {
3         List items = new ArrayList();
4         for (int j = 0; j < chunkSize; j++) {
5             Object item = itemReader.read();
6             Object processedItem = itemProcessor.process(item);
7             items.add(processedItem);
8         }
9         itemWriter.write(items);
10    }
11 }
```

Chunk 프로세싱 코드 흐름

```
1 public void Chunk_처리_방법(int chunkSize, int totalSize) {
2       for (int i = 0; i < totalSize; i = i + chunkSize) {
3         List items = new ArrayList();
4         for (int j = 0; j < chunkSize; j++) {
5             Object item = itemReader.read();
6             Object processedItem = itemProcessor.process(item);
7             items.add(processedItem);
8           }
9         itemWriter.write(items);
10    }
11 }
```

Chunk 프로세싱 코드 흐름

```
1 public void Chunk_처리_방법(int chunkSize, int totalSize) {
2     for (int i = 0; i < totalSize; i = i + chunkSize) {
3         List items = new ArrayList();
4         for (int j = 0; j < chunkSize; j++) {
5             Object item = itemReader.read();
6             Object processedItem = itemProcessor.process(item);
7             items.add(processedItem);
8           }
9         itemWriter.write(items);
10    }
11 }
```

Chunk 프로세싱 코드 흐름

```
1 public void Chunk_처리_방법(int chunkSize, int totalSize) {
2     for (int i = 0; i < totalSize; i = i + chunkSize) {
3         List items = new ArrayList(); // Reader에서 가져옴
4         for (int j = 0; j < chunkSize; j++) {
5             Object item = itemReader.read();
6             Object processedItem = itemProcessor.process(item);
7             items.add(processedItem);
8           }
9         itemWriter.write(items);
10    }
11 }
```

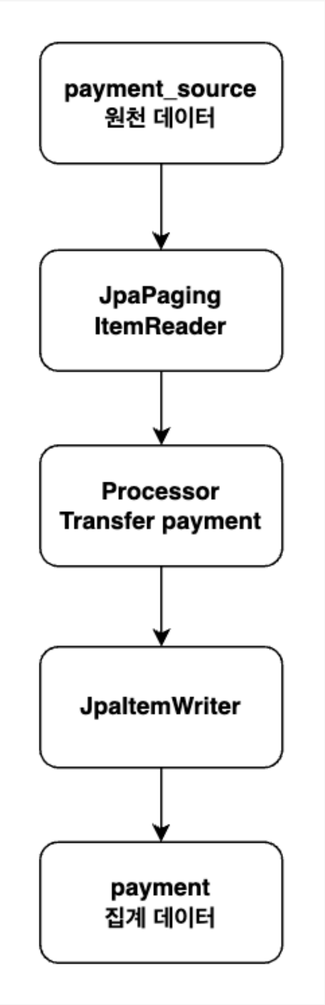
Chunk 프로세싱 코드 흐름

```
1 public void Chunk_처리_방법(int chunkSize, int totalSize) {
2       for (int i = 0; i < totalSize; i = i + chunkSize) {
3         List items = new ArrayList();
4         for (int j = 0; j < chunkSize; j++) {
5             Object item = itemReader.read();
6             Object processedItem = itemProcessor.process(item);
7             items.add(processedItem);
8           }
9         itemWriter.write(items);
10    }
11 }
```

Chunk 사이즈 조정, 주의해야 할 점

- 데이터베이스 트랜잭션 커밋 문제
  - Chunk가 커질수록 한 번에 커밋되는 데이터가 많아짐
  - 대량 커밋 시 DB Lock, Deadlock, Undo/Redo 로그 폭증 등으로 오히려 전체 성능 저하
  - 장애 발생 시 롤백 비용 커짐
- 외부 서비스 연동 시 네트워크 한계
  - 한 번에 대용량 데이터를 외부 시스템에 전송하면 상대 서버가 수신 거부, 타임아웃 발생 가능
  - 네트워크 대역폭을 초과해 실패하거나, 처리 지연
- 메모리 사용량 폭증
  - Chunk 사이즈가 클수록 한 번에 처리할 데이터를 메모리에 모두 올려야 함
  - JVM OutOfMemory 등, 서버 메모리 한계 초과로 배치 실패 위험
- 각 시스템 환경에 맞는 튜닝 필요
  - DB, 네트워크, 메모리 등 시스템 환경마다 최적값 다름
  - 실무에서는 보통 2,000 이상의 Chunk는 피하는 편
  - 직접 실험 및 모니터링을 통해 적정 Chunk Size 산정 필요

Chunk Reader-Processor-Writer 실전 예제



1. 데이터 추출 (Extract):
   - Reader가 PaymentSource 테이블에서 데이터를 Chunk Size 만큼 읽어온다.
   - JpaPagingItemReader을 활용한 Offset Limit 방식으로 조회
2. 데이터 가공 (Transform):
   - Processor에서 PaymentSource 데이터를 Payment 데이터로 가공한다.
3. 데이터 적재 (Load):
   - Writer가 가공된 Payment 데이터를 Payment 테이블에 저장한다.
   - JpaItemWriter 기반으로 insert

ItemReader

- ItemReader란?
  - Step에서 데이터를 읽어오는 역할
  - 다양한 데이터 소스(JPA, JDBC, 파일, API 등)에서 데이터 추출
- 주요 구현체와 특징
  - JpaPagingItemReader
    - JPA 기반, 페이징 처리로 대용량 데이터 안전하게 읽기
    - JPQL 쿼리 사용, 한 번에 메모리 과부하 방지
    - 주의: flush/clear 타이밍, 영속성 컨텍스트 관리 필요
  - JdbcCursorItemReader
    - DB Cursor 기반, 한 줄씩 직접 읽어오기
    - 주의: 커넥션 점유, 트랜잭션 길어질 경우 부하 발생
  - FlatFileItemReader
    - CSV/텍스트 파일 등 외부 파일 데이터 읽기
    - 주의: 파일 인코딩, 구분자, 헤더/스킵 처리 정확히 확인

ItemReader

- ItemReader 사용할 때 주의할 점
  - 대량 데이터 처리 시 한 번에 너무 많은 데이터 읽으면 OutOfMemory 등 메모리 이슈 발생
- DB 부하
  - 커서 방식 Reader는 DB에 부하, 커넥션 장시간 점유
  - 페이징 Reader는 쿼리 튜닝 필수, 인덱스 유무 확인
- 파일 Reader
  - 파일 용량, 포맷, 인코딩 이슈
  - 잘못된 포맷, 줄바꿈 문자 등으로 오류 발생 가능

ItemProcessor

- ItemProcessor란?
  - Reader에서 읽은 데이터를 가공, 변환, 필터링(제외)하는 역할
  - 입력값을 받아 가공 결과를 반환하거나, 필요 없는 데이터는 null로 반환해 Writer 단계로 전달하지 않음
- 주요 사용 사례
  - 데이터 포맷 변경(예: 날짜 포맷, 문자열 변환)
  - 검증/필터링(조건에 맞지 않는 데이터 제외)
  - 도메인 변환(엔티티 → DTO, 혹은 반대)
  - 복잡한 비즈니스 로직 반영
- ItemProcessor 사용할 때 주의할 점
  - 무거운 연산 주의: 외부 서비스 호출, 복잡한 연산이 많으면 전체 배치 성능 저하
  - 데이터 정합성: 필터링/가공 후 Writer에 넘기는 데이터와 실제 저장될 데이터의 정합성 항상 확인
  - null 반환 시 주의: Processor가 null을 반환하면 Writer로 넘어가지 않음(=필터링), 로직 오류 방지

ItemWriter

- ItemWriter란?
  - Processor에서 넘어온 데이터를 실제로 저장(쓰기)하는 역할 (DB, 파일, 외부 시스템 등 다양한 대상에 저장 가능)
- 주요 구현체와 특징
  - JpaItemWriter, JdbcBatchItemWriter
    - DB에 일괄 저장, 대량 데이터 처리에 적합
    - 주의: 트랜잭션, 커밋 타이밍, 영속성 컨텍스트 관리
  - FlatFileItemWriter
    - 파일(CSV, 텍스트 등)로 저장, 대량 로그/리포트 파일 생성에 유용
    - 주의: 파일 포맷, 인코딩, 줄바꿈 문자 등 체크
  - Custom Writer
    - 외부 API, 메일, 메시지 큐 등 다양한 방식 구현 가능
    - 주의: 외부 시스템 장애, 네트워크 타임아웃 등 예외처리 필수

ItemWriter

- ItemWriter 사용할 때 주의할 점
  - Chunk Size 영향: Writer는 한 번에 Chunk만큼의 데이터를 저장 (너무 크면 트랜잭션 부담, 너무 작으면 I/O 오버헤드)
  - 예외처리: 저장 실패 시 배치 전체가 중단될 수 있으니 롤백, 재시도 등 전략 필요
  - 실제 데이터 확인: 저장 후 데이터의 실제 상태(중복, 누락 등) 꼭 검증

예외 처리 및 리스너 활용

Fault-Tolerant(내결함성)

- 내결함성(Fault Tolerance)이란 무엇인가?
  - 정의
    - 배치 처리 중 작은 데이터 오류나 일시적 장애가 발생해도
    - Step 전체를 곧바로 실패시키지 않고, 건너뛰기(Skip) 또는 재시도(Retry) 로직을 적용해 계속 진행할 수 있도록 하는 기능
- 왜 필요한가?
  - 대용량 데이터 처리 시 일부 데이터 오류나 외부 시스템 장애가 항상 발생할 수 있음
  - 전체 Job이 단 한 건의 오류로 멈춘다면 생산성·가용성이 크게 저하
  - Fault-Tolerant를 통해 **신뢰성(resilience)**과 지속 **가용성(availability)** 확보

Fault-Tolerant 설정 메서드 정리

메서드	기능 분류	설명
.faultTolerant()	내결함성 활성화	Skip/Retry/BackOff 등의 내결함성 옵션을 사용할 수 있도록 StepBuilder를 FaultTolerantStepBuilder로 전환합니다.
.skip(Class<? extends Throwable> type)	Skip 설정	지정한 예외 타입이 발생했을 때 해당 아이템을 건너뛰도록 등록합니다.
.skipLimit(int)	Skip 설정	Skip 허용 최대 건수를 설정합니다. 이 횟수를 초과하면 Step이 실패합니다.
.skipPolicy(SkipPolicy)	Skip 설정	사용자 정의 또는 기본 SkipPolicy를 등록해, 복합 기준(예: 예외 타입 + 횟수)을 적용할 수 있습니다.
.noSkip(Class<? extends Throwable>)	Skip 설정	Skip 대상에서 제외할 예외 타입을 지정합니다.
.retry(Class<? extends Throwable>)	Retry 설정	지정한 예외 타입이 발생했을 때 재시도를 수행하도록 등록합니다.
.retryLimit(int)	Retry 설정	Retry 허용 최대 횟수를 설정합니다. 이 횟수를 초과하면 재시도를 중단합니다.
.retryPolicy(RetryPolicy)	Retry 설정	사용자 정의 또는 기본 RetryPolicy를 등록해, 복합 기준(예: 예외 타입 + 횟수)을 적용할 수 있습니다.
.backOffPolicy(BackOffPolicy)	Retry 설정	재시도 간격(고정/지수 등)을 조정하는 BackOff 정책을 등록합니다.
.noRetry(Class<? extends Throwable> type)	Retry 설정	Retry 대상에서 제외할 예외 타입을 지정합니다.
.noRollback(Class<? extends Throwable> type>)	롤백 제어	지정한 예외 타입이 발생해도 트랜잭션을 롤백하지 않고 커밋된 데이터를 유지합니다.

Fault-Tolerant 설정 메서드 정리

```
return new StepBuilder( name: "faultTolerantStep", jobRepository)
        .<PaymentSource, Payment>chunk( chunkSize: 100,
         transactionManager) SimpleStepBuilder<PaymentSource, Payment>
        .reader(paymentReader())
        .processor(paymentProcessor())
        .writer(paymentWriter())
        // 내결함성 활성화
        .faultTolerant() FaultTolerantStepBuilder<PaymentSource, Payment>

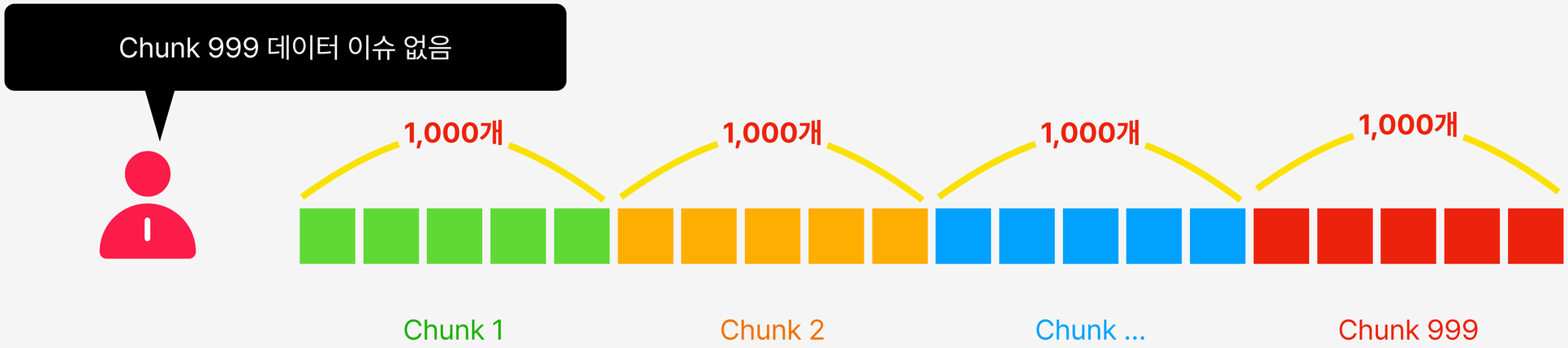
        // Skip 설정
        .skip( type: IllegalArgumentException.class)
        .skip( type: IllegalStateException.class)
        .skipLimit( skipLimit: 10) // 최대 10개까지 건너뛰기
        .noSkip( type: NullPointerException.class) // NullPointerException은 건너뛰지 않음
        .skipPolicy(customSkipPolicy()) // 커스텀 스킵 정책 적용

        // Retry 설정
        .retry( type: DeadlockLoserDataAccessException.class)
        .retry( type: OptimisticLockingFailureException.class)
        .retry( type: QueryTimeoutException.class)
        .retryLimit( retryLimit: 3) // 최대 3번 재시도
        .retryPolicy(retryPolicy()) // 커스텀 재시도 정책 적용
        .backOffPolicy(backOffPolicy()) // 재시도 간격 정책
        .noRetry( type: IllegalArgumentException.class) // IllegalArgumentException은 재시도하지 않음

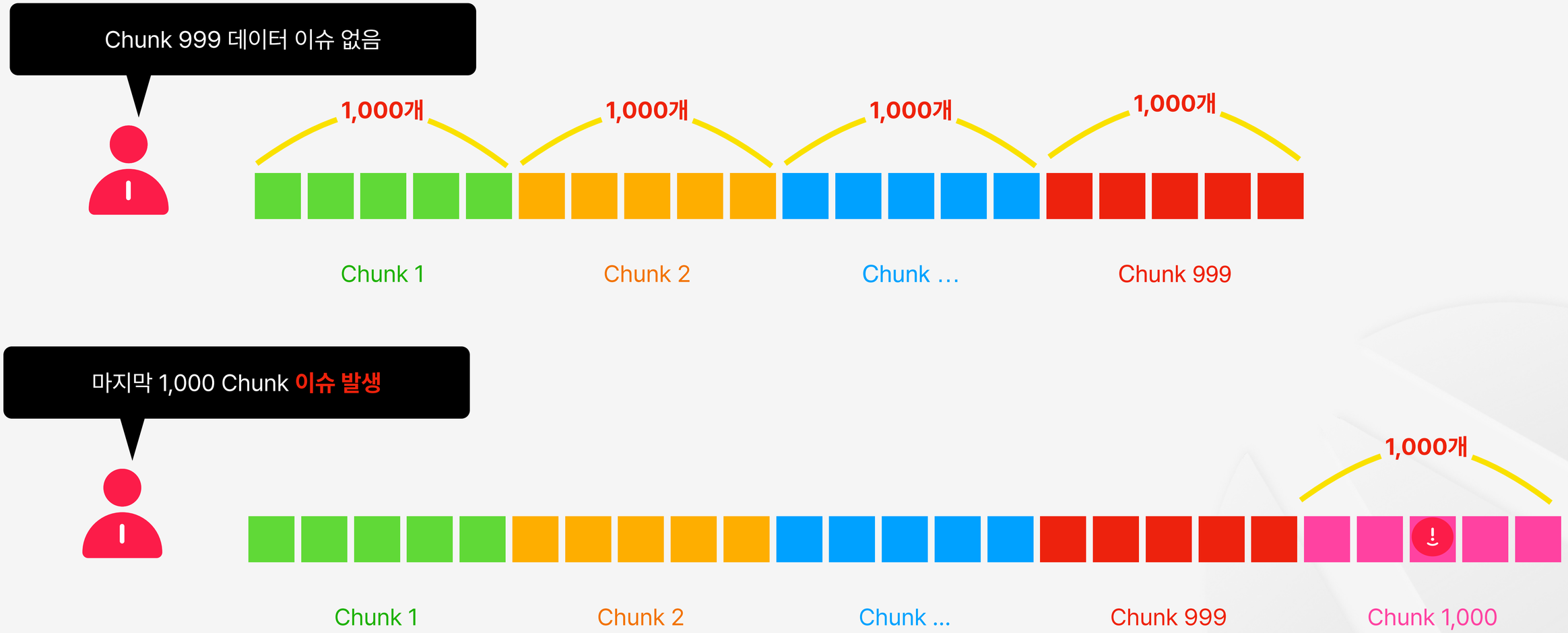
        // 롤백 제어
        .noRollback( type: IllegalStateException.class) // IllegalStateException 발생 시 롤백하지 않음

        .build();
```

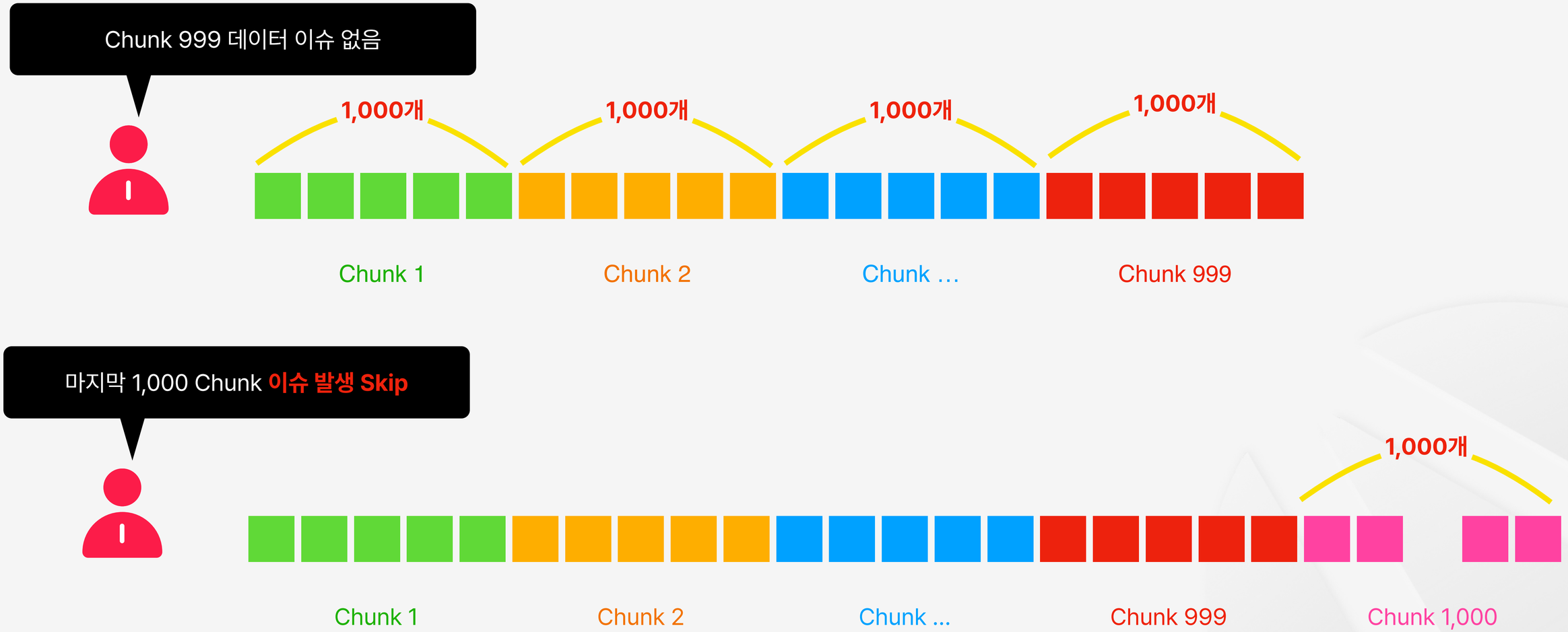
Skip 시나리오: “영구적으로 손상된 데이터 건너뛰기”



Skip 시나리오: “영구적으로 손상된 데이터 건너뛰기”



Skip 시나리오: “영구적으로 손상된 데이터 건너뛰기”



Skip 시나리오: “영구적으로 손상된 데이터 건너뛰기”

- 배경
  - 주문 데이터(예: CSV 파일) 한 줄씩 읽어서 처리하는 Job.
  - 비즈니스 룰상 금액(amount), 날짜(date), 고객 ID 등 필수 필드가 올바른 포맷과 값 범위를 가져야 함.
- 문제
  - 파일에 한 줄이 잘못된 포맷(예: 2025-13-01 같은 불가능한 날짜) 혹은 음수 금액으로 잘못 들어가 있음.
  - 이 데이터는 무조건 수동으로 교정해야 할 ‘영구적 오류’이므로, 재시도해도 절대 올바르게 변환되지 않음.
- 해결: Skip
  - .skip(InvalidFormatException.class).skipLimit(100)
  - 최대 100건까지 이런 오류 라인을 건너뛰며 나머지 정상 데이터를 계속 처리
  - 왜 Skip?
    - 잘못된 데이터는 몇 번 재시도해도 복구 불가능 → 재시도 의미 없음
    - 일일이 Job을 중단하고 재실행하는 번거로움 대신, 오류 건만 빠르게 제외하고 나머지를 처리

Skip 시나리오: “영구적으로 손상된 데이터 건너뛰기”

- Skip(건너뛰기) 기능
  - **skip()**: 특정 예외 발생 시 해당 아이템을 건너뜁니다.
  - **skipLimit()**: 최대 건너뛰기 횟수를 설정합니다.
  - **noSkip()**: 특정 예외는 건너뛰지 않고 Step을 실패시킵니다.
  - **skipPolicy()**: 커스텀 스킵 정책을 적용합니다.
- SkipPolicy 제공 클래스
  - **LimitCheckingItemSkipPolicy**: 특정 예외 타입에 대해 지정된 횟수까지만 스킵을 허용합니다.
  - **AlwaysSkipItemSkipPolicy**: 항상 스킵을 수행합니다.
  - **NeverSkipItemSkipPolicy**: 항상 스킵을 수행하지 않습니다.

LimitCheckingItemSkipPolicy

- 역할
  - 지정된 예외 타입일 때만, 그리고 총 스킵 횟수가 `skipLimit`을 초과하지 않을 때만 스킵을 허용하는 기본 정책.
- 동작 방식
  - 매 예외 발생 시 현재까지의 skipCount와 비교하여 `skipCount < skipLimit` 이면 스킵(`true`), 초과 시 `false`.
- 설정 예시
  -
```
new StepBuilder("faultTolerantStep", jobRepository)
      .<I, O>chunk(10)
      .reader(reader()).processor(proc()).writer(writer())
      .faultTolerant()
      .skip(MyBusinessException.class)     // 스킵 허용 예외 등록
      .skipLimit(5)                        // 최대 5건 스킵
      .build();
```
- 장점/용도
  - 비즈니스 예외에 유용, **횟수 기반**으로 안전하게 스킵 제어
  - 대부분의 간단한 skip 시나리오에 적합

AlwaysSkipItemSkipPolicy

- 역할
  - 모든 예외를 무조건 스킵하도록 허용하는 정책.
- 동작 방식
  - shouldSkip(...) 호출 시 예외 타입이나 발생 횟수와 상관없이 항상 true 반환.
- 설정 예시
  - ```
    new StepBuilder("faultTolerantStep", jobRepository)
        .<I, O>chunk(10)
        .reader(reader()).processor(proc()).writer(writer())
        .faultTolerant()
        .skipPolicy(new AlwaysSkipItemSkipPolicy())
        .build();
    ```
- 장점/용도
  - **모든 오류 무시** 하고 끝까지 처리하고 싶을 때
  - 개발 초기 테스트 나 PoC 단계에서 **오류를 잠시 억제**
    ```

NeverSkipItemSkipPolicy

- 역할
  - 어떤 예외도 스킵하지 않는 정책.
- 동작 방식
  - shouldSkip(...) 호출 시 언제나 false를 반환 → 예외 발생 시 Step 즉시 실패(롤백)
- 설정 예시
  -
```
new StepBuilder("faultTolerantStep", jobRepository)
    .<I, O>chunk(10)
    .reader(reader()).processor(proc()).writer(writer())
    .faultTolerant()
    .skipPolicy(new NeverSkipItemSkipPolicy())
    .build();
```
- 장점/용도
  - 스킵 기능을 완전히 비활성화하고, 오류 발생 즉시 중단하고 싶을 때
  - 가장 엄격한 오류 처리 시나리오

내결함성 활성화 & Skip 설정 코드

```
return new StepBuilder( name: "paymentReportStep", jobRepository)
        .<~>chunk( chunkSize: 20, transactionManager) SimpleStepBuilder<PaymentSource, Payment>
        .reader(paymentReportReader)
        .processor(itemProcessor())
        .writer(paymentReportWriter())
        .faultTolerant() FaultTolerantStepBuilder<PaymentSource, Payment>
        .skipLimit( skipLimit: 2) // 최대 2번까지 skip 허용, skip = 예외 발생
        .skip( type: InvalidPaymentAmountException.class)
          .skipPolicy(new LimitCheckingItemSkipPolicy()) // Limit 횟수 만큼 skip을 허용, 초과시 예외 발생
          .skipPolicy(new AlwaysSkipItemSkipPolicy()) // 항상 skip을 하는 정책
          // 항상 skip을 허용하지 않음, 다음 Policy 예를 들어 LimitCheckingItemSkipPolicy가 동작
        .skipPolicy( skipPolicy: new NeverSkipItemSkipPolicy())
        .build();
```

createSkipPolicy() 내부: 기본 정책 선택

```
protected SkipPolicy createSkipPolicy() {
    SkipPolicy skipPolicy = this.skipPolicy;
    Map<Class<? extends Throwable>, Boolean> map = new HashMap<>( m: skippableExceptionClasses);
    map.put(ForceRollbackForWriteSkipException.class, true);
    LimitCheckingItemSkipPolicy limitCheckingItemSkipPolicy = new LimitCheckingItemSkipPolicy(skipLimit, skippableExceptions: map);
    if (skipPolicy == null) {
        if (skippableExceptionClasses.isEmpty() && skipLimit > 0) {
            logger.debug(String.format(
                    "A skip limit of %s is set but no skippable exceptions are defined. Consider defining skippable exceptions.",
                    skipLimit));
        }
        skipPolicy = limitCheckingItemSkipPolicy;
    }
    else if (limitCheckingItemSkipPolicy != null) {
        skipPolicy = new CompositeSkipPolicy( skipPolicies: new SkipPolicy[]{skipPolicy, limitCheckingItemSkipPolicy});
    }
    return skipPolicy;
}
```

createSkipPolicy() 내부: 기본 정책 선택

```
protected SkipPolicy createSkipPolicy() {
    SkipPolicy skipPolicy = this.skipPolicy;
    Map<Class<? extends Throwable>, Boolean> map = new HashMap<>( m: skippableExceptionClasses);
    map.put(ForceRollbackForWriteSkipException.class, true);
    LimitCheckingItemSkipPolicy limitCheckingItemSkipPolicy = new LimitCheckingItemSkipPolicy(skipLimit, skippableExceptions: map);
    if (skipPolicy == null) {
        if (skippableExceptionClasses.isEmpty() && skipLimit > 0) {
            logger.debug(String.format(
                    "A skip limit of %s is set but no skippable exceptions are defined. Consider defining skippable exceptions.",
                    skipLimit));
        }
        skipPolicy = limitCheckingItemSkipPolicy;
    }
    else if (limitCheckingItemSkipPolicy != null) {
        skipPolicy = new CompositeSkipPolicy( skipPolicies: new SkipPolicy[]{skipPolicy, limitCheckingItemSkipPolicy});
    }
    return skipPolicy;
}
```

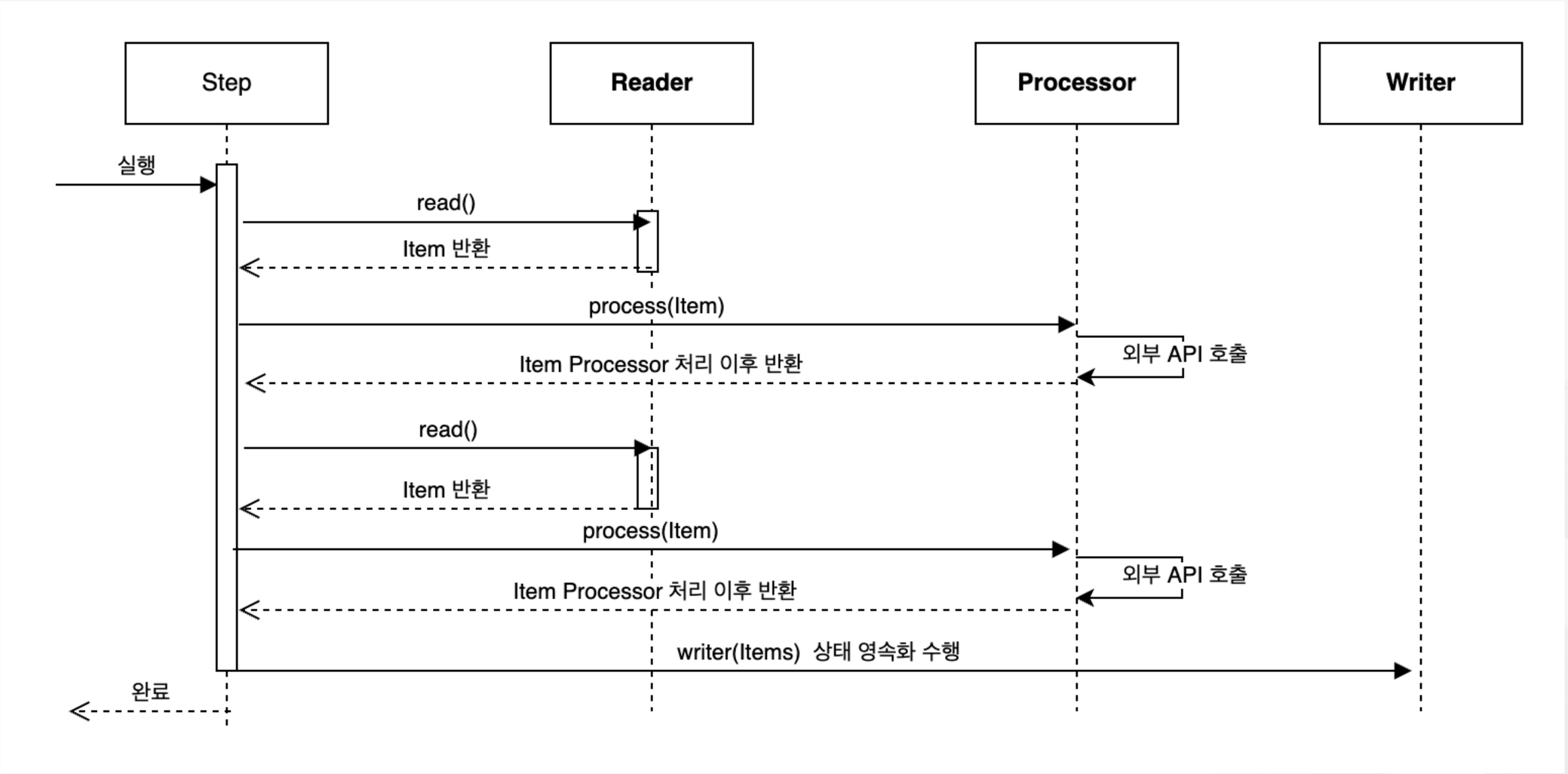
CompositeSkipPolicy#shouldSkip 순차 검사

```
@Override
public boolean shouldSkip(Throwable t, long skipCount) throws SkipLimitExceededException {
    for (SkipPolicy policy : skipPolicies) {  policy: NeverSkipItemSkipPolicy@9541    skipP
        if (policy.shouldSkip( t: t, skipCount)) {  skipCount: 0    policy: NeverSkipItemSkip
            return true;
        }
    }
    return false;
}
```

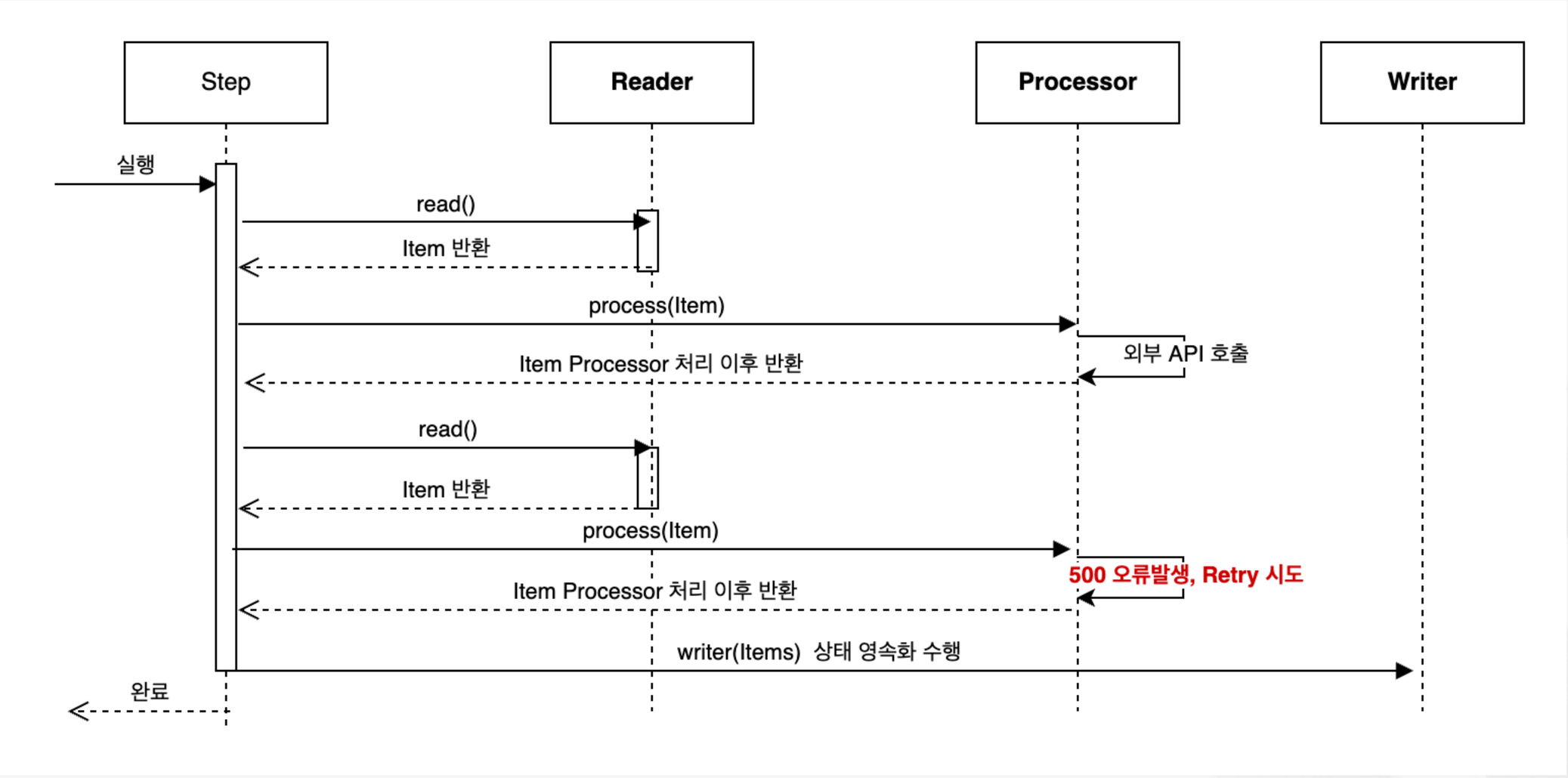
최종 LimitCheckingItemSkipPolicy 적용

```java
@Override
public boolean shouldSkip(Throwable t, long skipCount) {
    if (skippableExceptionClassifier.classify( classifiable: t)) {
        if (skipCount < skipLimit) {
            return true;
        }
        else {
            throw new SkipLimitExceededException(skipLimit, t);
        }
    }
    else {
        return false;
    }
}
```

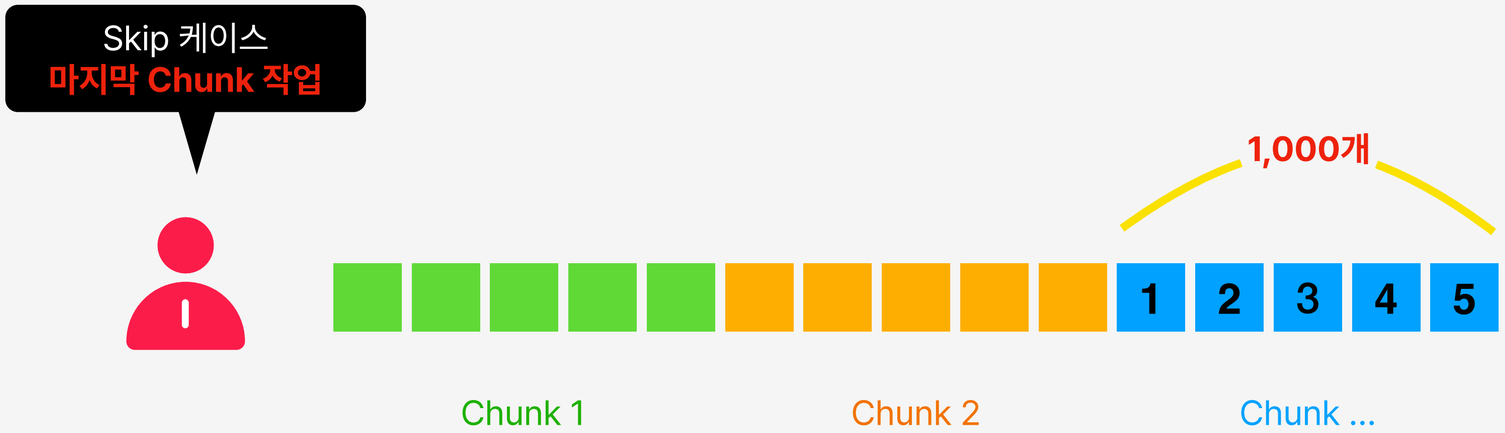
Retry 시나리오: “일시적 장애에 대한 자동 재시도”



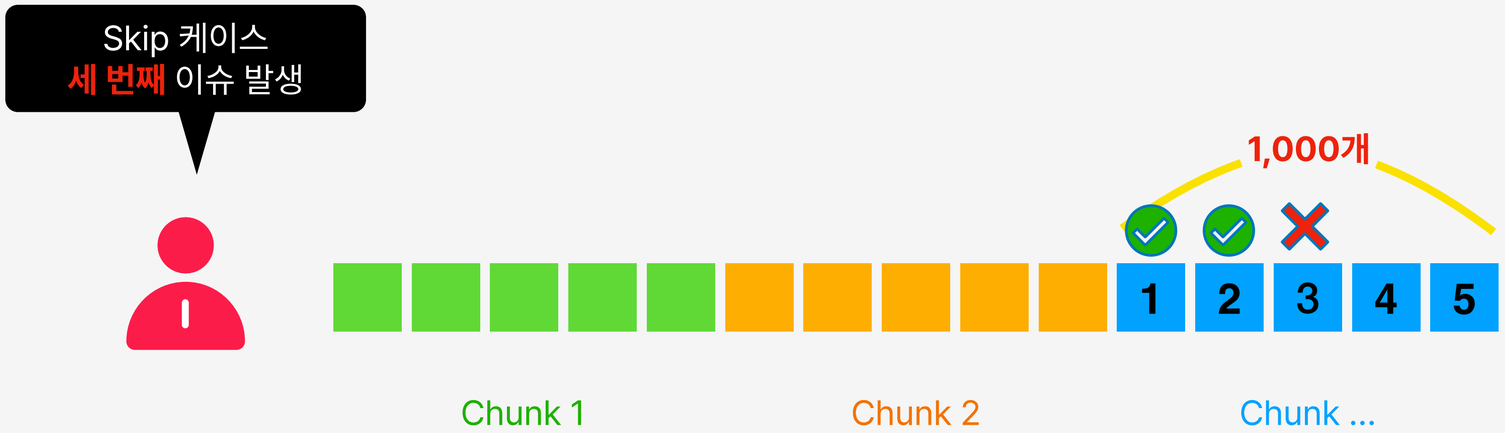
Retry 시나리오: “일시적 장애에 대한 자동 재시도”



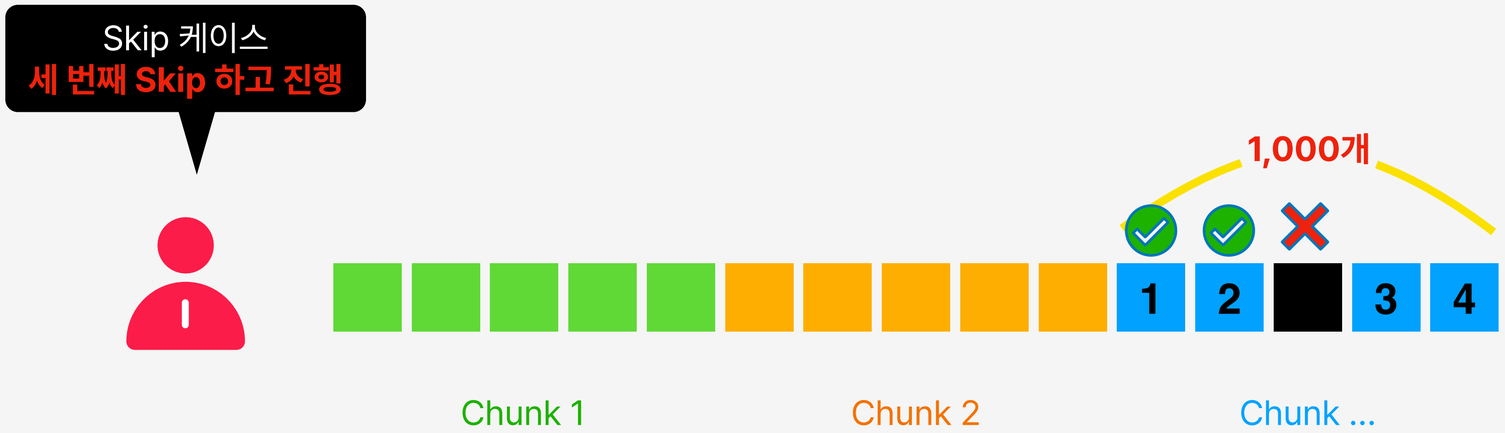
Retry 시나리오: “일시적 장애에 대한 자동 재시도”



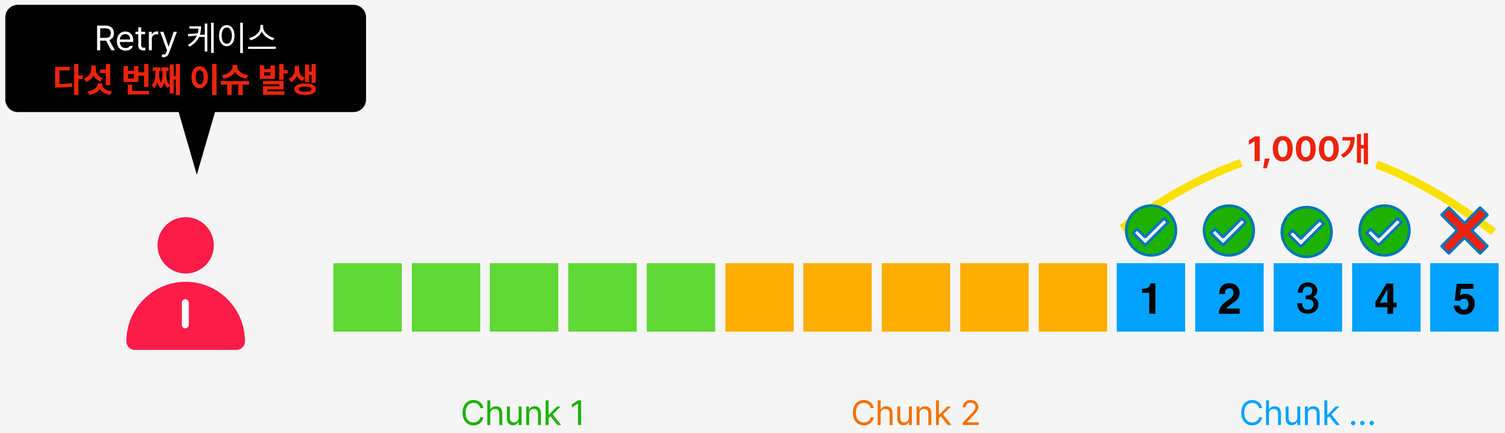
Retry 시나리오: “일시적 장애에 대한 자동 재시도”



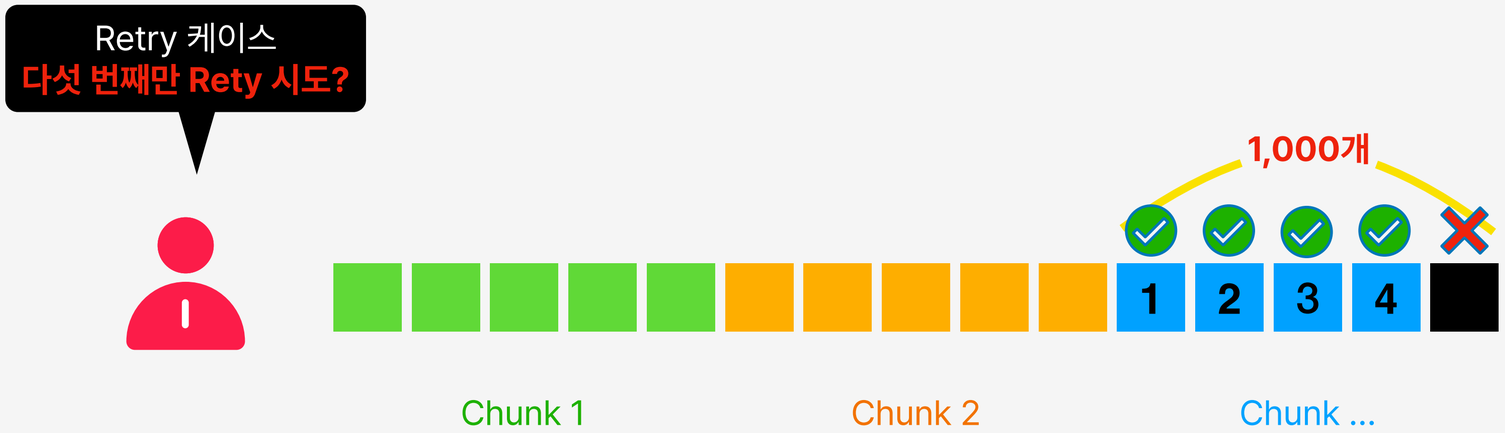
Retry 시나리오: “일시적 장애에 대한 자동 재시도”



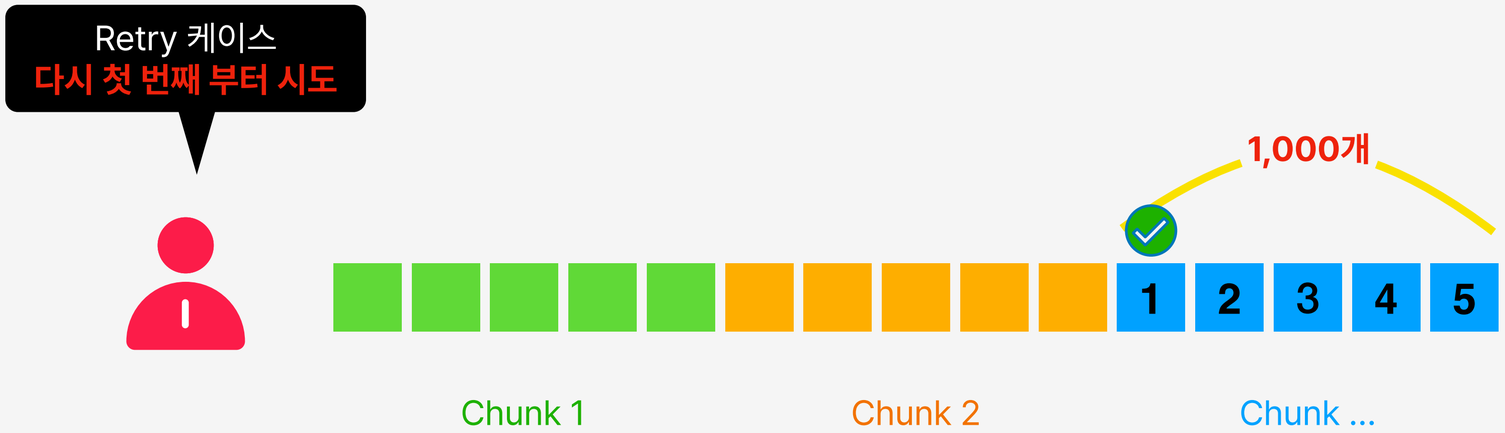
Retry 시나리오: “일시적 장애에 대한 자동 재시도”



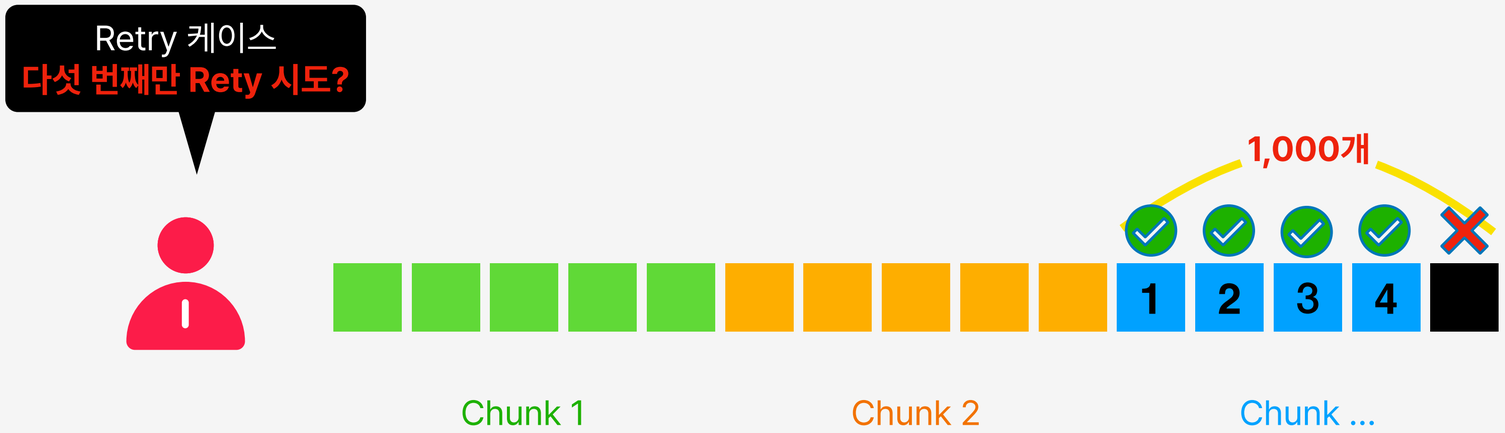
Retry 시나리오: “일시적 장애에 대한 자동 재시도”



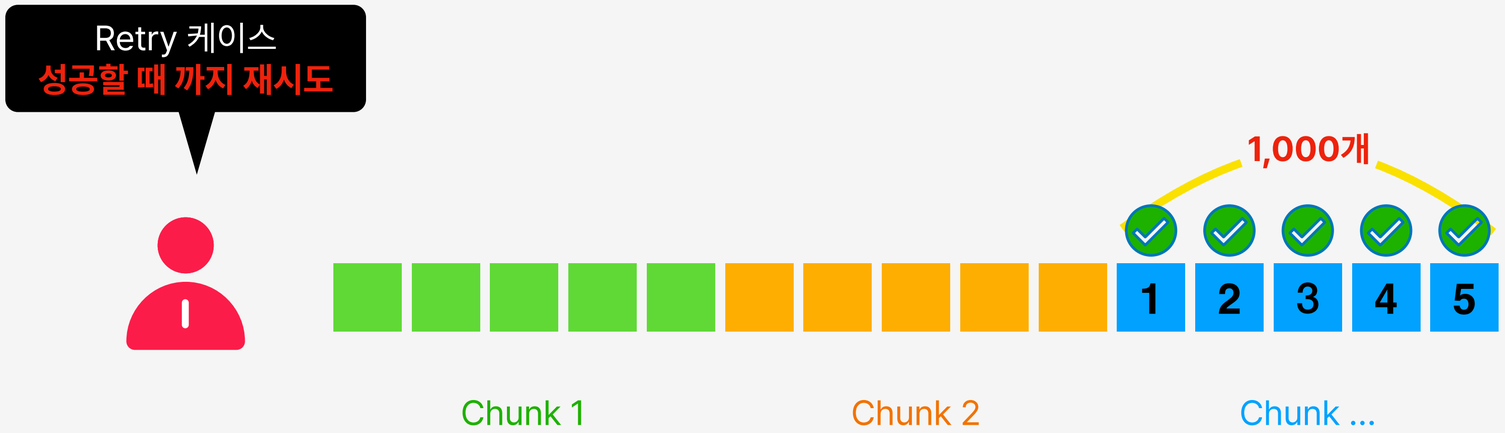
Retry 시나리오: “일시적 장애에 대한 자동 재시도”



Retry 시나리오: “일시적 장애에 대한 자동 재시도”



Retry 시나리오: “일시적 장애에 대한 자동 재시도”



Retry 시나리오: “일시적 장애에 대한 자동 재시도”

- 배경
  - 결제 처리 Job에서, 처리된 내역을 외부 결제 API 또는 다른 DB 테이블에 기록해야 함.
- 문제
  - 네트워크 일시 장애나 DB Deadlock 등의 TransientError 발생
  - 첫 번째 시도에서만 실패하고, 잠시 기다렸다가 다시 요청하면 성공할 가능성이 높음
- 해결: Retry
  - .retry(DeadlockLoserDataAccessException.class).retryLimit(3).backOffPolicy(new FixedBackOffPolicy(1000))
  - 최대 3회(초기 포함) 재시도, 재시도 사이 1초 지연
  - 재시도 중에 API 요청이 성공하면 나머지 처리를 계속함
- 왜 Retry 해야하는가?
  - 장애가 영구적이 아니라 일시적이므로, 재시도로 대부분 해결
  - Job 전체를 중단하지 않고, 자동으로 회복해 연속성 확보

Retry 시나리오: “일시적 장애에 대한 자동 재시도”

- 재시도 설정 주요 메서드
  - **retry(Class<? extends Throwable> type) :** 지정한 예외가 발생하면 재시도를 수행하도록 등록합니다.
  - **retryLimit(int retryLimit) :** 최대 재시도 횟수를 설정합니다. 이 횟수를 초과하면 더 이상 재시도하지 않습니다.
  - **noRetry(Class<? extends Throwable> type)** : 재시도 대상에서 제외할 예외 타입을 지정합니다.
  - **retryPolicy(RetryPolicy)** : 사용자 정의 또는 기본 제공 RetryPolicy를 등록해, 예외 타입·횟수·기타 조건으로 재시도를 세밀하게 제어할 수 있습니다.
  - **backOffPolicy(BackOffPolicy)** : 재시도 사이의 간격(고정/지수 등)을 조정하는 BackOffPolicy를 등록합니다.
- RetryPolicy 제공 클래스
  - **SimpleRetryPolicy** : 지정한 예외 타입에 대해 최대 횟수(maxAttempts)만큼 재시도하는 가장 기본적인 정책
  - **TimeoutRetryPolicy** : 지정된 기간(timeout) 동안만 재시도를 허용하는 정책
  - **AlwaysRetryPolicy** : 어떤 예외든 제한 없이 재시도하는 정책 (canRetry()가 항상 true)
  - **NeverRetryPolicy**: 어떤 예외든 재시도하지 않는 정책 (canRetry()가 항상 false)
  - **CompositeRetryPolicy:** 여러 RetryPolicy를 조합하여, 예외 종류나 상황에 따라 순차적으로 정책을 적용

SimpleRetryPolicy

- 역할
  - 지정된 예외 타입에 대해 **최대 횟수 기반**으로 재시도를 허용하는 기본 정책.
- 동작 방식
  - 내부에 Map<Class<? extends Throwable>, Boolean>을 설정해, 어떤 예외를 재시도할지 등록.
  - 재시도 횟수가 retryLimit을 초과하지 않았으면 canRetry(...)가 true를 반환.
- 설정 예시

```
Map<Class<? extends Throwable>, Boolean> retryable = Map.of(
    DeadlockLoserDataAccessException.class, true,
    IllegalArgumentException.class, false
);
RetryPolicy policy = new SimpleRetryPolicy(3, retryable, true);

new StepBuilder("retryStep", jobRepository)
  .<I, O>chunk(10)
  .reader(reader()).processor(proc()).writer(writer())
  .faultTolerant()
  .retryPolicy(policy)
  .build();
```

  - 세 번(3)까지 DeadlockLoserDataAccessException에 대한 재시도를 허용하고, IllegalArgumentException은 즉시 실패.
- 장점/용도
  - 횟수 기반 가장 간단한 재시도 정책
  - “이 예외만, 최대 N번” 같은 기본 시나리오에 적합.

TimeoutRetryPolicy

- 역할
  - 지정된 시간(timeout) 동안만 재시도를 허용하는 정책.
- 동작 방식
  - 첫 시도가 시작된 시점을 기준으로 누적 경과시간을 체크.
  - 경과시간이 timeout 미만이면 open(...) 이후 canRetry(...)가 true.
- 설정 예시
  -
```
new StepBuilder("retryStep", jobRepository)
  .<I, O>chunk(10)
  .reader(reader()).processor(proc()).writer(writer())
  .faultTolerant()
  // 5초 동안 재시도 허용
  .retryPolicy(new TimeoutRetryPolicy(5_000))
  .build();
```
  - 5초 이내라면 동일 예외가 계속 발생해도 재시도.
- 장점/용도
  - “몇 초 안에 해결될 일시적 오류”에 적합.
  - 네트워크 지연, 외부 API 응답 대기 같은 시나리오에서 유용.

AlwaysRetryPolicy

- 역할
  - 모든 예외를 **무조건 재시도**하도록 허용하는 정책.
- 동작 방식
  - `canRetry(...)` 호출 시 발생 횟수나 예외 타입에 상관없이 **항상 `true`** 반환.
- 설정 예시

```
new StepBuilder("retryStep", jobRepository)
  .<I, O>chunk(10)
  .reader(reader()).processor(proc()).writer(writer())
  .faultTolerant()
  .retryPolicy(new AlwaysRetryPolicy())
  .build();
```

  - 어떠한 예외도 재시도 없이 바로 Step 실패 처리.
- 장점/용도
  - PoC 단계에서 “일단 다 재시도해 보자” 할 때
  - 실무에서는 잘 쓰지 않음

NeverRetryPolicy

- 역할
  - 전혀 재시도하지 않음을 선언하는 정책.
- 동작 방식
  - canRetry(...)가 항상 false → 예외 발생 즉시 실패(롤백).
- 설정 예시

```
new StepBuilder("retryStep", jobRepository)
  .<I, O>chunk(10)
  .reader(reader()).processor(proc()).writer(writer())
  .faultTolerant()
  .retryPolicy(new NeverRetryPolicy())
  .build();
```

  - 어떠한 예외도 재시도 없이 바로 Step 실패 처리.
- 장점/용도
  - 재시도를 완전히 비활성화하고 싶을 때.
  - 재시도 자체가 비즈니스에 부정적일 때(예: 중복 처리나 일관성 문제)

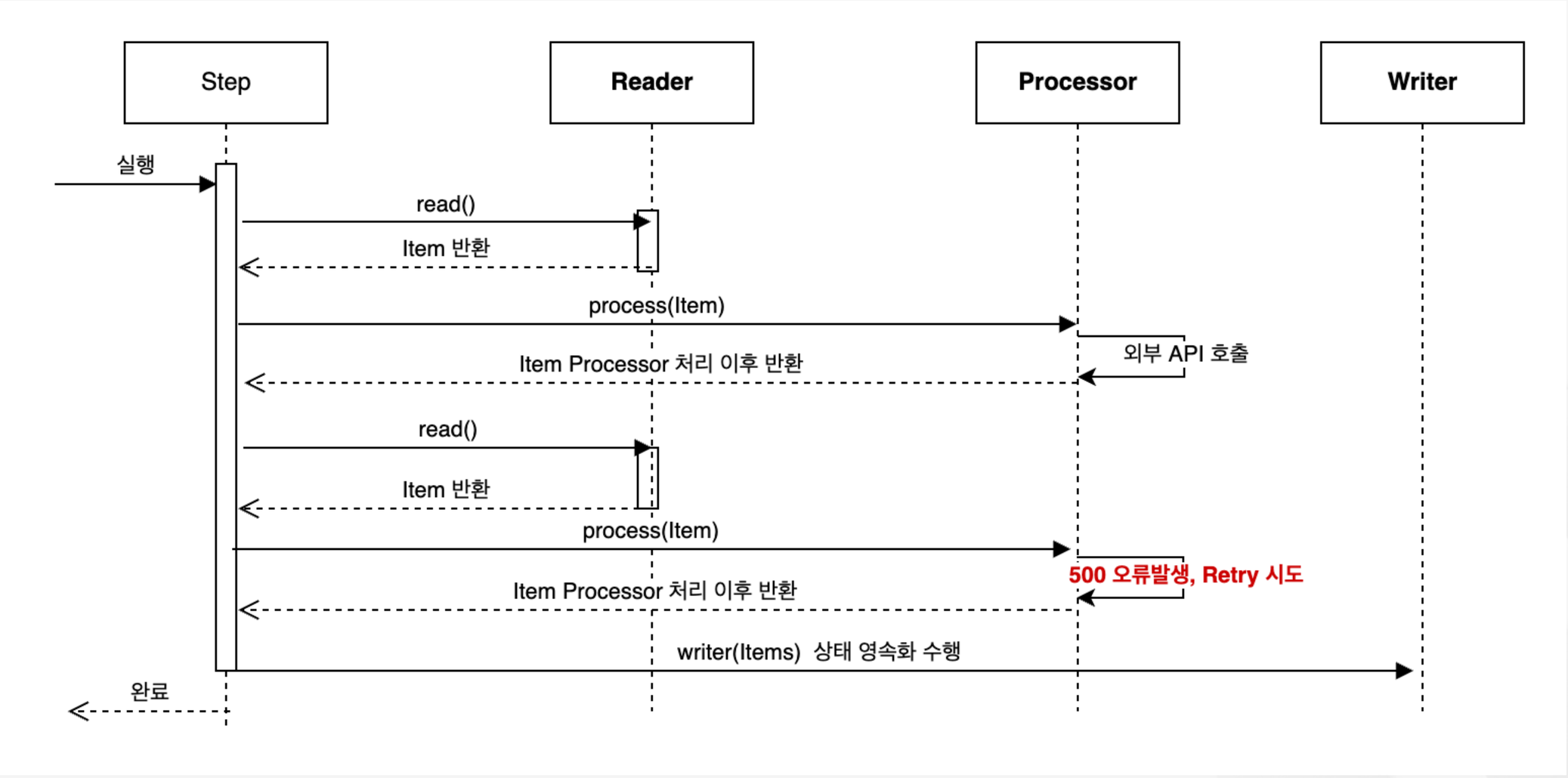
CompositeRetryPolicy

- 역할
  - 여러 개의 RetryPolicy를 묶어서, 예외 발생 시 정책을 순차적으로 적용하는 복합 재시도 정책입니다.
  - 예를 들어, 사용자가 직접 지정한 RetryPolicy와 Spring Batch가 기본 생성하는 SimpleRetryPolicy를 함께 조합할 수 있습니다.
- 동작 방식
  - canRetry(context) 호출 시, 순서대로 각 정책의 canRetry(...)를 검사 하나라도 true를 반환하면 즉시 true 리턴 → 재시도 허용, 모두 false일 때만 false 리턴 → 재시도 중단
- 설정 예시
-

```
if (retryPolicy == null) {
    Assert.state( expression: !(retryableExceptionClasses.isEmpty() && retryLimit > 0),
            message: "If a retry limit is provided then retryable exceptions must also be specified");
    retryPolicy = simpleRetryPolicy;
}
else if ((!retryableExceptionClasses.isEmpty() && retryLimit > 0)) {   retryLimit: 10    retryableExceptio
    CompositeRetryPolicy compositeRetryPolicy = new CompositeRetryPolicy();   compositeRetryPolicy: Compos
    compositeRetryPolicy.setPolicies(new RetryPolicy[]{retryPolicy, simpleRetryPolicy});   simpleRetryPoli
    retryPolicy = compositeRetryPolicy;   retryPolicy: TimeoutRetryPolicy@8983    compositeRetryPolicy: Co
}
```

- 장점/용도
  - 다양한 재시도 규칙을 한 번에 적용 가능, 복잡한 운영 환경에서 정교한 장애 대응 시나리오로 확장하기에 적합합니다.
  - 사용자가 .retryPolicy(...)로 커스텀 정책을 주입하더라도, Spring Batch가 기본 생성하는 SimpleRetryPolicy를 그대로 유지해 누락 없이 예외를 처리

RetryPolicy 실습 시나리오



createRetryOperations() 내부: 기본 정책 선택

```
protected BatchRetryTemplate createRetryOperations() {

    RetryPolicy retryPolicy = this.retryPolicy;   retryPolicy: TimeoutRetryPolicy@8982    retryPoli
    SimpleRetryPolicy simpleRetryPolicy = null;   simpleRetryPolicy: "SimpleRetryPolicy[maxAttempts

    Map<Class<? extends Throwable>, Boolean> map = new HashMap<>( m: retryableExceptionClasses);   m
    map.put(ForceRollbackForWriteSkipException.class, true);
    simpleRetryPolicy = new SimpleRetryPolicy( maxAttempts: retryLimit,  retryableExceptions: map);   map:

    if (retryPolicy == null) {
        Assert.state( expression: !(retryableExceptionClasses.isEmpty() && retryLimit > 0),
                message: "If a retry limit is provided then retryable exceptions must also be speci
        retryPolicy = simpleRetryPolicy;
    }
    else if ((!retryableExceptionClasses.isEmpty() && retryLimit > 0)) {   retryLimit: 10    retrya
        CompositeRetryPolicy compositeRetryPolicy = new CompositeRetryPolicy();   compositeRetryPol
        compositeRetryPolicy.setPolicies(new RetryPolicy[]{retryPolicy, simpleRetryPolicy});   simp
        retryPolicy = compositeRetryPolicy;   retryPolicy: TimeoutRetryPolicy@8982    compositeRetr
    }
```

createRetryOperations() 내부: 기본 정책 선택

```
protected BatchRetryTemplate createRetryOperations() {

    RetryPolicy retryPolicy = this.retryPolicy;   retryPolicy: TimeoutRetryPolicy@8982    retryPoli
    SimpleRetryPolicy simpleRetryPolicy = null;   simpleRetryPolicy: "SimpleRetryPolicy[maxAttempts

    Map<Class<? extends Throwable>, Boolean> map = new HashMap<>( m: retryableExceptionClasses);   m
    map.put(ForceRollbackForWriteSkipException.class, true);
    simpleRetryPolicy = new SimpleRetryPolicy( maxAttempts: retryLimit,  retryableExceptions: map);   map:

    if (retryPolicy == null) {
        Assert.state( expression: !(retryableExceptionClasses.isEmpty() && retryLimit > 0),
                message: "If a retry limit is provided then retryable exceptions must also be speci
        retryPolicy = simpleRetryPolicy;
    }
    else if ((!retryableExceptionClasses.isEmpty() && retryLimit > 0)) {   retryLimit: 10    retrya
        CompositeRetryPolicy compositeRetryPolicy = new CompositeRetryPolicy();   compositeRetryPol
        compositeRetryPolicy.setPolicies(new RetryPolicy[]{retryPolicy, simpleRetryPolicy});   simp
        retryPolicy = compositeRetryPolicy;   retryPolicy: TimeoutRetryPolicy@8982    compositeRetr
    }
```

CompositeRetryPolicy#canRetry 검사

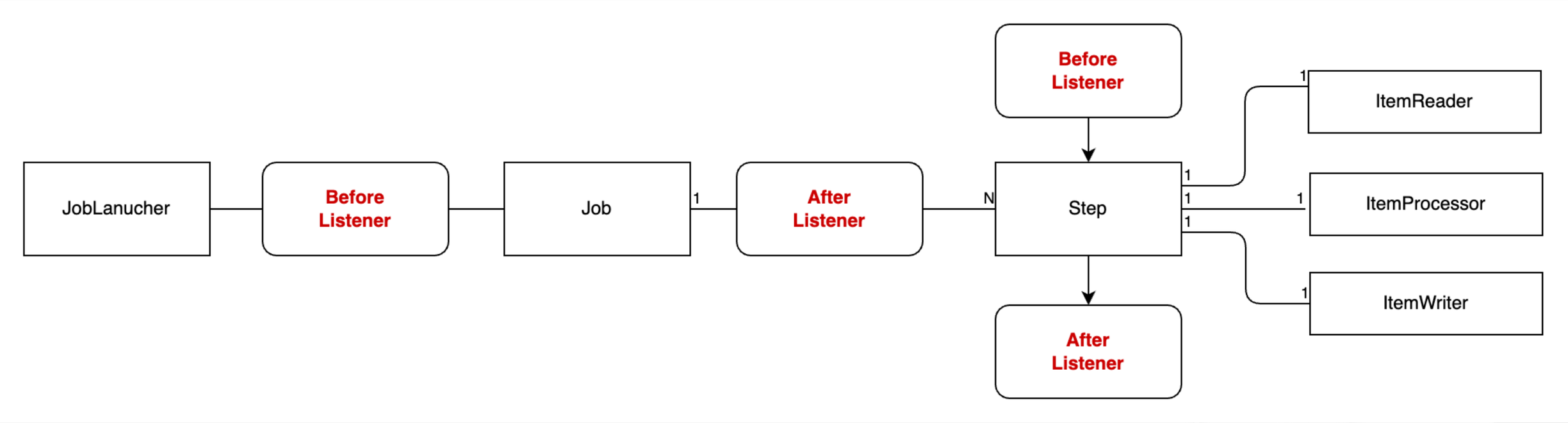
```
public boolean canRetry(RetryContext context) {
    else {
        for (int i = 0; i < contexts.length; i++) {
            if (!policies[i].canRetry( context: contexts[i])) {
                retryable = false;
            }
        }
    }

    return retryable;
}
```

SimpleRetryPolicy#canRetry 검사

```
@Override
public boolean canRetry(RetryContext context) {
    Throwable t = context.getLastThrowable();
    boolean can = (t == null || retryForException(t)) && context.getRetryCount() < getMaxAttempts();
    if (!can && t != null && !this.recoverableClassifier.classify( classifiable: t)) {
        context.setAttribute( name: RetryContext.NO_RECOVERY,  value: true);
    }
    else {
        context.removeAttribute( name: RetryContext.NO_RECOVERY);
    }
    return can;
}
```

Listener


JobLanucher
Before Listener
Job
1
After Listener
N
Step
Before Listener
After Listener
1
1
1
1
1
1
ItemReader
ItemProcessor
ItemWriter

JobExecutionListener

```
public interface JobExecutionListener {

    Callback before a job executes.
    Params: jobExecution – the current JobExecution
    default void beforeJob(JobExecution jobExecution) {
    }

    Callback after completion of a job. Called after both
    successful and failed executions. To perform logic on a
    particular status, use if (jobExecution.getStatus()
    == BatchStatus.X).
    Params: jobExecution – the current JobExecution
    default void afterJob(JobExecution jobExecution) {
    }

}
```

- 언제 호출?
  - beforeJob(JobExecution) → 잡 시작 직전
  - afterJob(JobExecution) → 모든 스텝 종료 후
- 주요 활용 포인트
  - 리소스 초기화/정리
    - ex) 외부 API 커넥션 열기 / 닫기
  - 전체 처리 요약 리포트
    - jobExecution.getStartTime()·getEndTime() 으로 총 수행 시간 계산
    - jobExecution.getStatus() 로 성공·실패 분기
    - 메타테이블(BATCH_JOB_EXECUTION) 로 파라미터·스텝 로그 집계
  - 알림/통지
    - 잡 시작 전 슬랙/메일 알림
    - 잡 완료 후 결과 보고

StepExecutionListener

```
public interface StepExecutionListener extends StepListener {

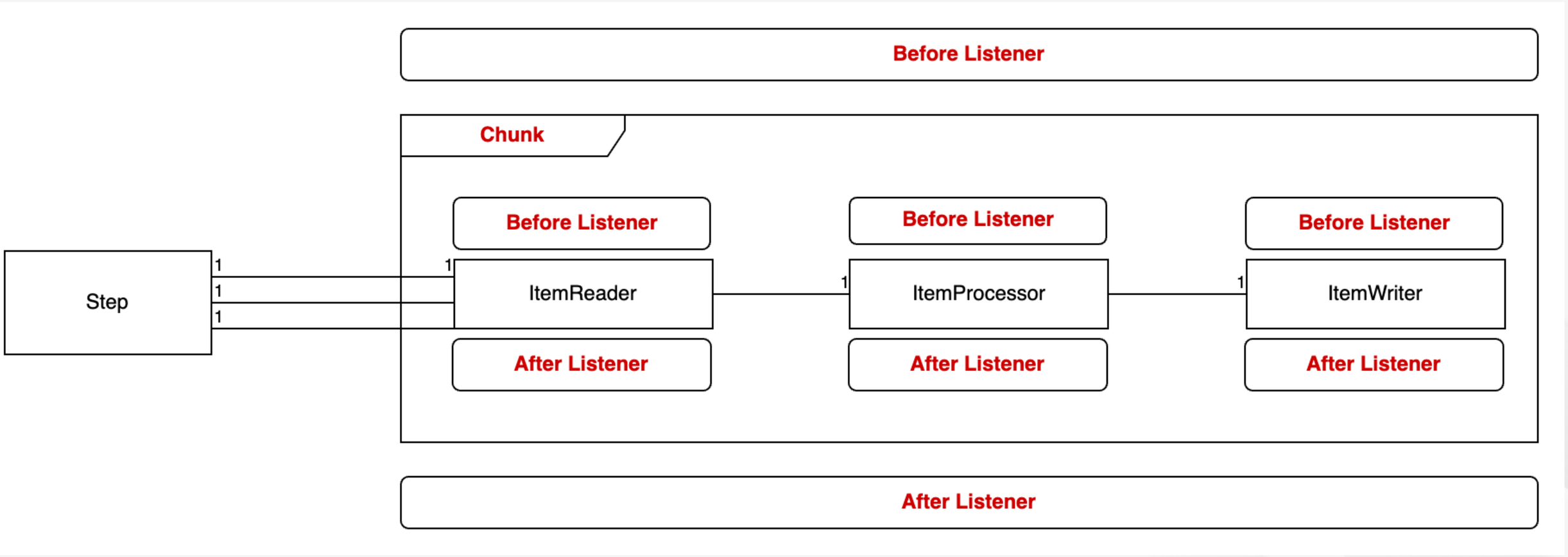
    Initialize the state of the listener with the StepExecution
    from the current scope.
    Params: stepExecution – instance of
            StepExecution.
    default void beforeStep(StepExecution stepExecution) {
    }

    Give a listener a chance to modify the exit status from a step.
    The value returned is combined with the normal exit status
    by using ExitStatus.and(ExitStatus).
    Called after execution of the step's processing logic
    (whether successful or failed). Throwing an exception in this
    method has no effect, as it is only logged.
    Params: stepExecution – a StepExecution instance.
    Returns: an ExitStatus to combine with the normal
             value. Return null (the default) to leave the old
             value unchanged.
    @Nullable
    default ExitStatus afterStep(StepExecution stepExecution) { return null; }

}
```

- 언제 호출?
  - beforeStep(StepExecution) → 각 스텝 시작 전
  - afterStep(StepExecution) → 각 스텝 종료 후
- 주요 활용 포인트
  - 단계별 타이밍 측정
    - beforeStep 에 타임스탬프 저장 → afterStep 에 차이 계산
  - 건별 처리 통계
    - getReadCount(), getWriteCount(), getSkipCount() 등 집계 → 로깅 또는 DB 저장
  - 조건부 흐름 제어
    - afterStep 반환 ExitStatus 변경 → 이후 잡 흐름에 반영
  - 글로벌 컨텍스트 설정
    - ExecutionContext 에 값 저장 → 다음 스텝에서 참조

Chunk 관련 Listener



ChunkListener

```java
public interface ChunkListener extends StepListener {

    The key for retrieving the rollback exception.
    String ROLLBACK_EXCEPTION_KEY = "sb_rollback_exception";

    Callback before the chunk is executed, but inside the
    transaction.
    Params: context – The current ChunkContext
    default void beforeChunk(ChunkContext context) {
    }

    Callback after the chunk is executed, outside the transaction.
    Params: context – The current ChunkContext
    default void afterChunk(ChunkContext context) {
    }

    Callback after a chunk has been marked for rollback. It is
    invoked after transaction rollback. While the rollback will
    have occurred, transactional resources might still be active
    and accessible. Due to this, data access code within this
    callback still "participates" in the original transaction unless it
    declares that it runs in its own transaction. As a result, you
    should use PROPAGATION_REQUIRES_NEW for any
    transactional operation that is called from here.
    Params: context – the chunk context containing the
            exception that caused the underlying rollback.
    default void afterChunkError(ChunkContext context) {
    }

}
```

- 언제 호출?
  - beforeChunk(ChunkContext)
    - 트랜잭션이 시작된 직후, 청크 단위 처리 직전에
  - afterChunk(ChunkContext)
    - 청크 처리가 성공적으로 커밋된 직후
  - afterChunkError(ChunkContext)
    - 청크 처리 중 예외가 발생해 롤백된 직후
- 주요 활용 포인트
  - 청크 경계 타이밍 측정
    - beforeChunk에서 타임스탬프 기록 → afterChunk에서 차이 계산
  - 트랜잭션 오류 처리
    - afterChunkError에서 롤백 원인 로깅, 알림 또는 별도 리커버리 로직 수행
- 처리 메트릭 집계
  - 성공한 청크 수, 실패된 청크 수 집계 → 모니터링 시스템에 전송

ItemReadListener

```java
public interface ItemReadListener<T> extends StepListener {

    /**
     * Called before {@link ItemReader#read()}
     */
    default void beforeRead() {
    }

    /**
     * Called after {@link ItemReader#read()}. This method is called only for actual items
     * (that is, it is not called when the reader returns {@code null}).
     *
     * Params: {@code item} – returned from read()
     */
    default void afterRead(T item) {
    }

    /**
     * Called if an error occurs while trying to read.
     *
     * Params: {@code ex} – thrown from {@link ItemReader}
     */
    default void onReadError(Exception ex) {
    }

}
```

- 언제 호출?
  - beforeRead() → ItemReader.read() 호출 직전
  - afterRead(T item) → read()가 정상적으로 객체를 반환한 직후
  - onReadError(Exception ex) → read() 도중 예외 발생 시 바로
- 주요 활용 포인트
  - 읽기 성능 모니터링
    - 각 레코드별 읽기 시간 측정 → 병목 구간 탐지
  - 데이터 품질 검사
    - afterRead에서 값 검증, 이상치 필터링 로직 삽입
  - 에러 대비 로깅/알림
    - onReadError에서 예외 정보 로깅 또는 관리자 알림

ItemProcessListener

```
public interface ItemProcessListener<T, S> extends StepListener {

    Called before ItemProcessor.process(Object).

    Params: item – to be processed.
    default void beforeProcess(T item) {
    }

    Called after ItemProcessor.process(Object) returns.
    If the processor returns null, this method is still called,
    with a null result, allowing for notification of "filtered"
    items.

    Params: item – to be processed
            result – of processing
    default void afterProcess(T item, @Nullable S result) {
    }

    Called if an exception was thrown from ItemProcessor.
    process(Object).

    Params: item – attempted to be processed
            e – - exception thrown during processing.
    default void onProcessError(T item, Exception e) {
    }

}
```

- 언제 호출?
  - beforeProcess(T item) → ItemProcessor.process(item) 호출 직전
  - afterProcess(T item, S result) → process()가 정상적으로 결과를 반환한 직후
  - onProcessError(T item, Exception e) → process() 도중 예외 발생 시
- 주요 활용 포인트
  - 프로세싱 타이밍 측정
    - 각 아이템별 처리 시간 기록 → CPU 집약 로직 최적화
  - 필터링 알림
    - afterProcess에서 result == null인 필터링된 아이템 로깅
  - 예외 세부 처리
    - onProcessError에서 특정 항목 재처리 또는 스킵 전략 결정
- 통계 집계
  - 성공/실패 처리 건수 집계 → 장애 패턴 분석

ItemWriteListener

```
public interface ItemWriteListener<S> extends StepListener {

    Called before ItemWriter.write(Chunk)

    Params: items – to be written
    default void beforeWrite(Chunk<? extends S> items) {
    }

    Called after ItemWriter.write(Chunk). This is called
    before any transaction is committed, and before
    ChunkListener.afterChunk(ChunkContext).

    Params: items – written items
    default void afterWrite(Chunk<? extends S> items) {
    }

    Called if an error occurs while trying to write. Called inside a
    transaction, but the transaction will normally be rolled back.
    There is no way to identify from this callback which of the
    items (if any) caused the error.

    Params: exception – thrown from ItemWriter
            items – attempted to be written.
    default void onWriteError(Exception exception, Chunk<? extends S> items) {
    }

}
```

- 언제 호출?
  - beforeWrite(List<? extends S> items) → ItemWriter.write(items) 호출 직전
  - afterWrite(List<? extends S> items) → write()가 정상적으로 커밋(랜잭션 커밋)된 이후
  - onWriteError(Exception exception, List<? extends S> items) → write() 도중 예외 발생 시
- 주요 활용 포인트
  - 쓰기 타이밍 측정
    - 쓰기 호출 전후 시간 기록 → DB I/O 최적화 방안 도출
  - 배치 커밋 조정
    - afterWrite에서 커밋 간격 조정 또는 배치 사이즈 재계산
  - 오류 아이템 로깅
    - onWriteError에서 실패한 레코드 상세 로깅 → 재처리 스크립트 활용
- 트랜잭션 프로퍼티 설정
  - 필요 시 PROPAGATION_REQUIRES_NEW 등을 통해 별도 트랜잭션 처리

Part 2 Spring Batch 성능 최적화 기법

01

스프링 배치 성능 개선 Reader

- Limit offset 방식
- Zero offset 방식
- Cursor 방식

02

스프링 배치 성능 개선 Processor & Writer

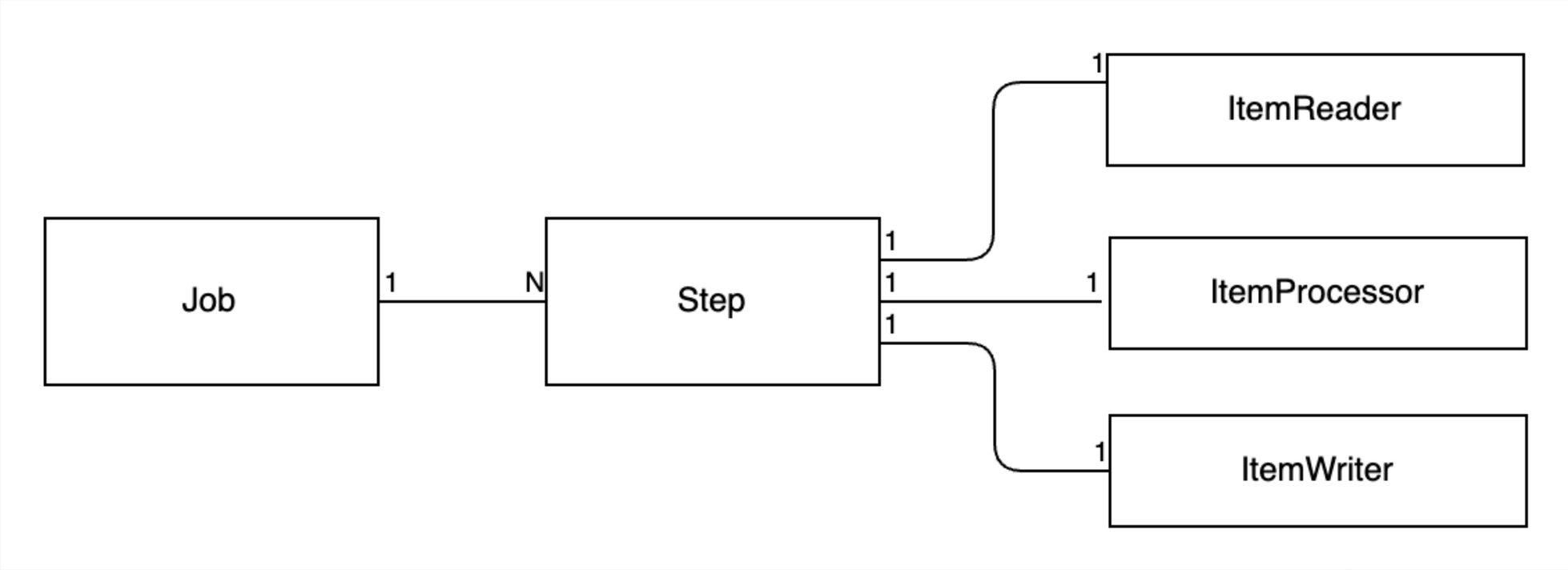
- Processor 성능 개선
- Writer 성능 개선
- 병목 원인 파악 및 개선

03

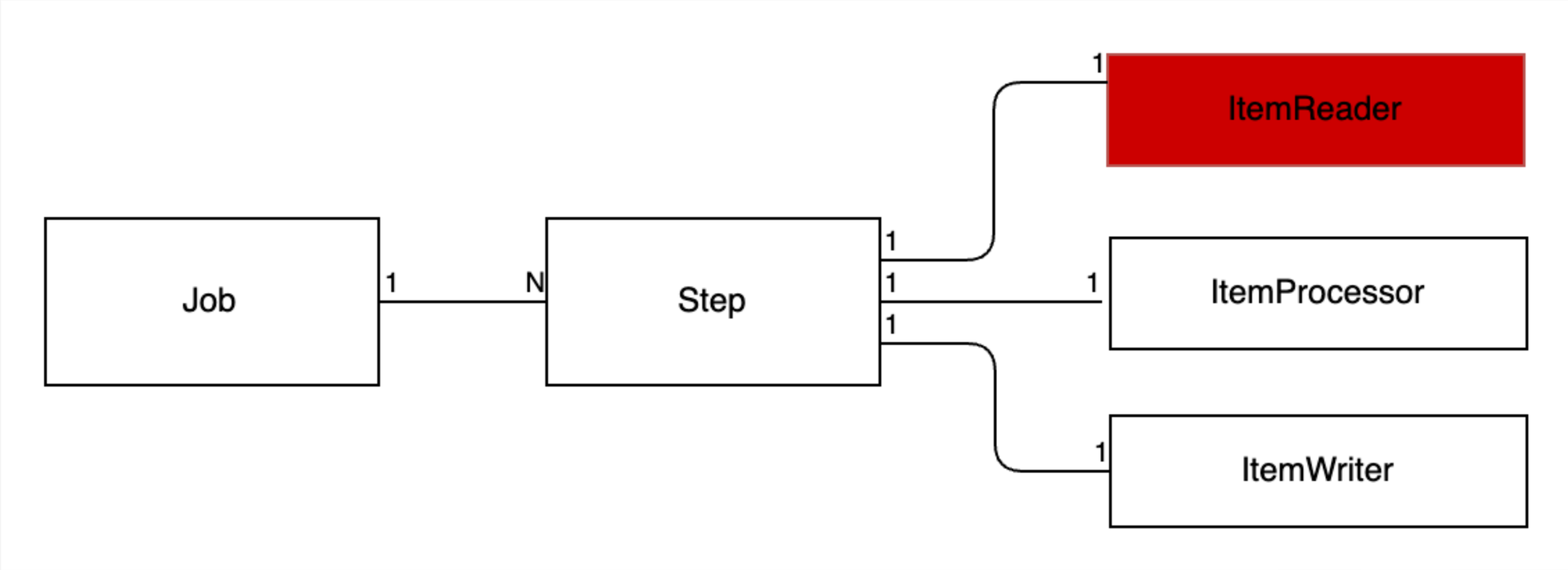
스프링 배치 성능 개선 Partition Step

- 병렬 Partition Step 처리
- MasterStep, Slave Step 수행 구조

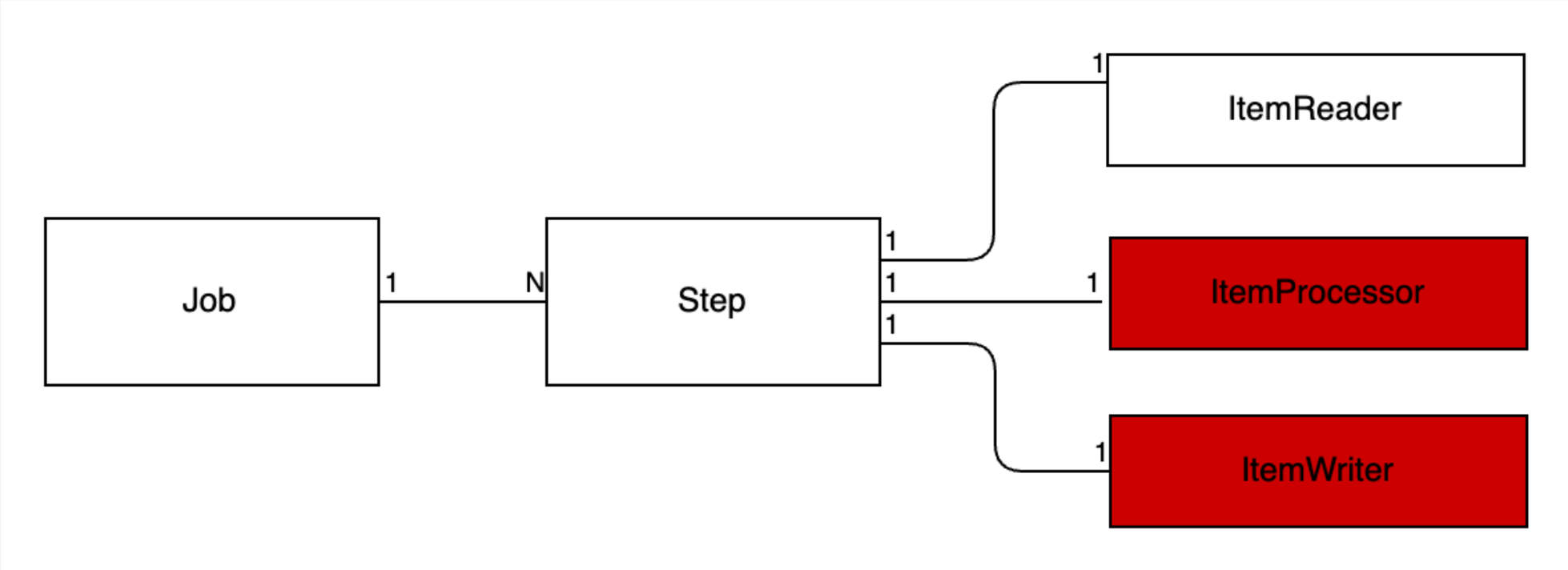
스프링 배치 성능 개선



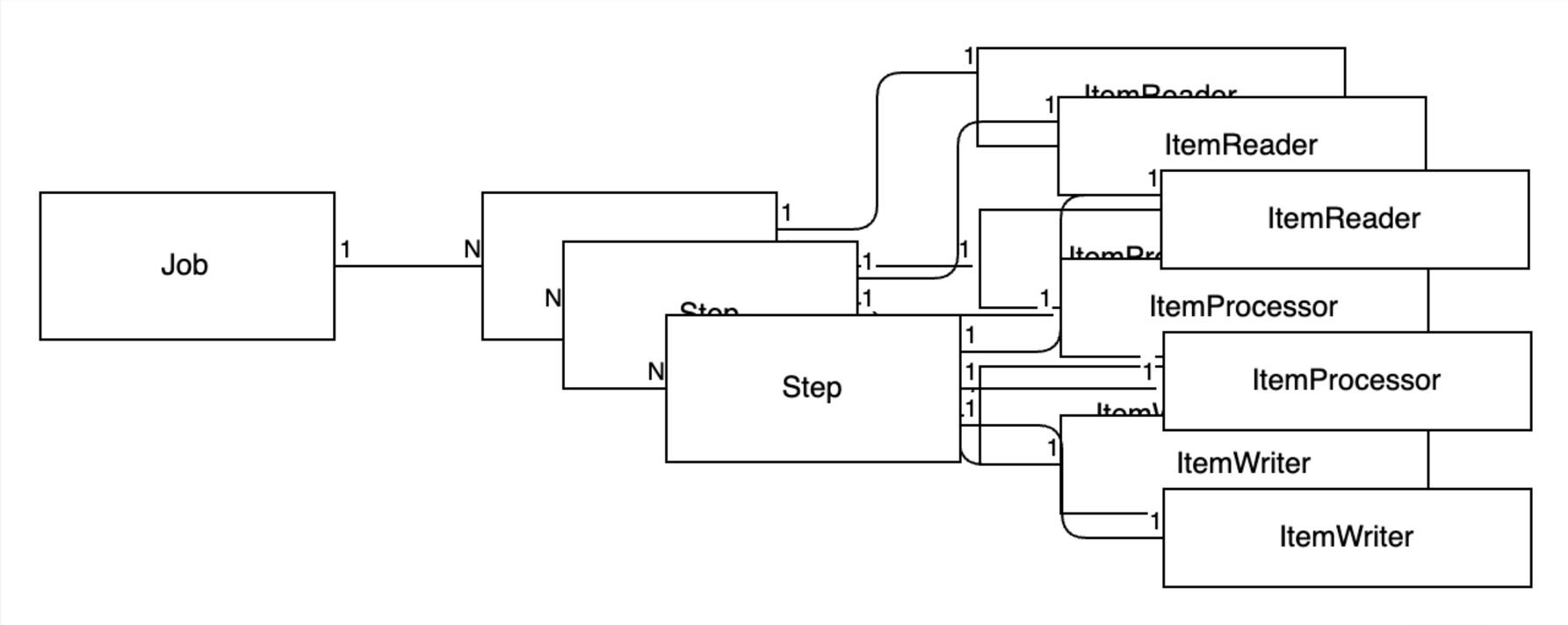
Reader 성능 개선



Processor & Writer 성능 개선



Partition Step 성능 개선



스프링 배치 Reader 성능 개선

01

Limit-Offset 방식

- OFFSET + LIMIT 으로 건너뛰기
- 구현 매우 간단
- 페이지가 뒤로 갈 수록 느려짐

02

No-Offset 방식

- WHERE lastKey>... + LIMIT (키셋)
- 인덱스 Seek 기반, 일관된 성능
- 애플리케이션에서 상태(마지막 키) 관리

03

Cursor 방식

- DB에 커서 열고 한 행씩 순차적으로 읽기
- JDBC 서버-사이드 커서(fetchSize)
- 스트리밍 읽기, 메모리 최소화
- 단일 스레드 & 커넥션 장기 점유

실습 Sample Data Setup

```
-- 기존 테이블 drop
drop table payment_source;

-- payment_source 테이블 생성
create table payment_source
(
    id                                  bigint auto_increment
        primary key,
    payment_date                        date           not null,
    discount_amount                     decimal(38, 2) not null,
    final_amount                        decimal(38, 2) not null,
    original_amount                     decimal(38, 2) not null,
    partner_business_registration_number varchar(100)  not null,
    partner_corp_name                   varchar(100)   not null
);

-- payment_date 컬럼에 대한 인덱스 생성 (조회 성능 향상)
create index idx_payment_date
    on payment_source (payment_date);
```

실습 Sample Data Setup

```
-- 10,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 10000);

-- 50,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 50000);

-- 100,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 100000);

-- 500,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 500000);

-- 1,000,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 1000000);

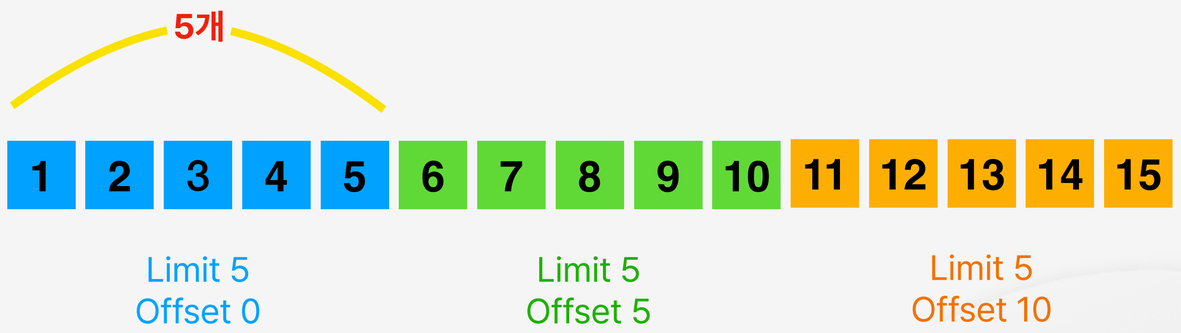
-- 5,000,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 5000000);
```

성능 측정 코드

```
@Bean
public Step paymentReportStep(
        JpaPagingItemReader<PaymentSource> paymentReportReader
) {
    return new StepBuilder( name: "paymentReportStep", jobRepository)
            .<~>chunk( chunkSize: 1_000, transactionManager)
            // Step 소요 시간 측정
            .listener( listener: new StepDurationTrackerListener())
            .reader(paymentReportReader)
            .processor(paymentReportProcessor())
            .writer(paymentReportWriter())
            // Chunk 소요 시간 측장
            .listener( listener: new ChunkDurationTrackerListener())
            .build();
```

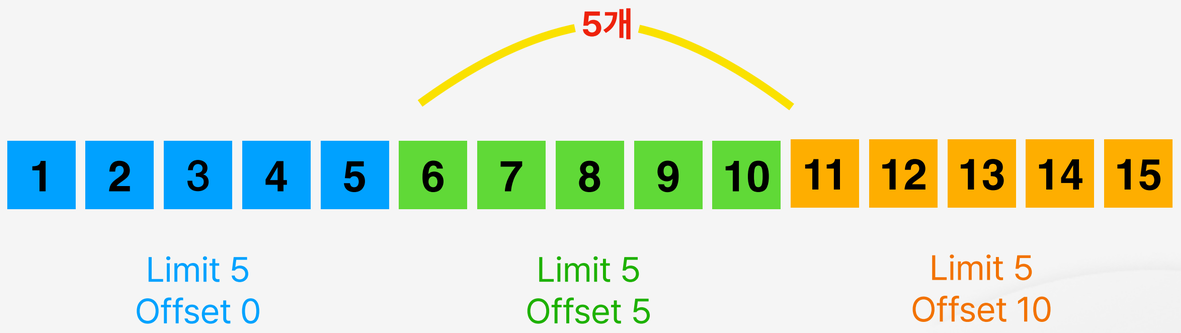
Limit-Offset 방식

```sql
select *
from payment_source
limit 5 offset 0
;
```



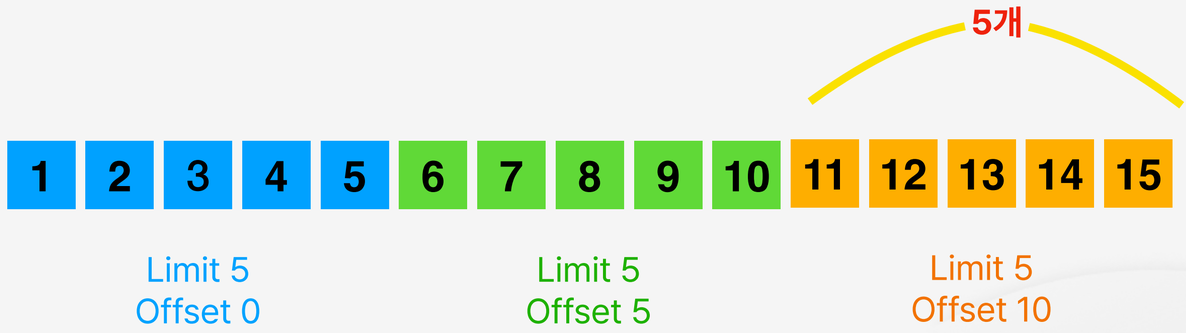
Limit-Offset 방식

```
select *
from payment_source
limit 5 offset 5
;
```



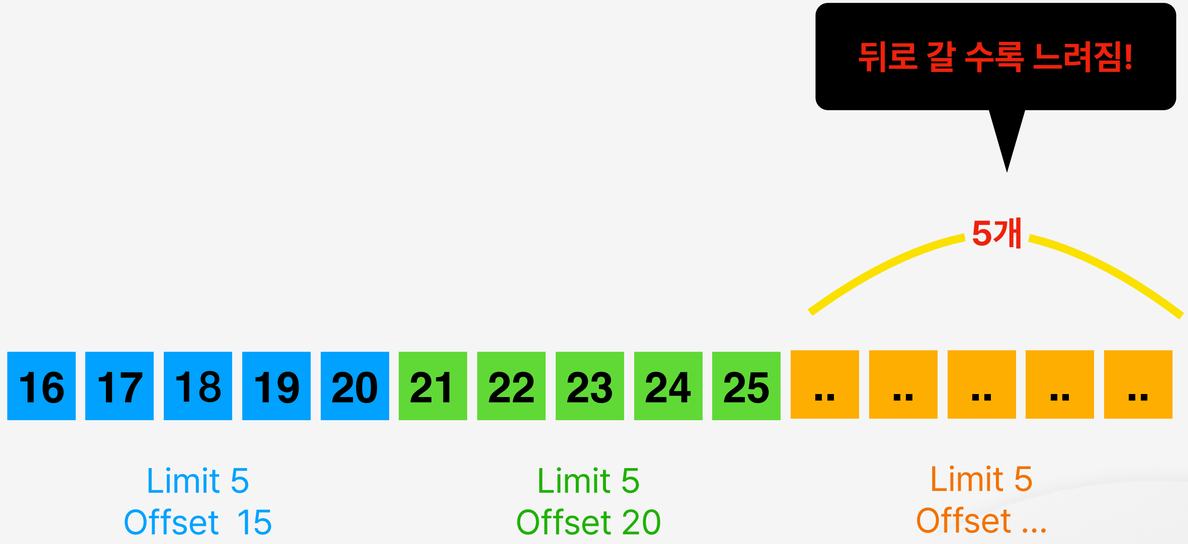
Limit-Offset 방식

```
select *
from payment_source
limit 5 offset 10
;
```



Limit-Offset 방식

```
select *
from payment_source
limit 5 offset 100000
;
```



실습 Sample Data Setup

```sql
-- 기존 테이블 drop
drop table payment_source;

-- payment_source 테이블 생성
create table payment_source
(
    id                                  bigint auto_increment
        primary key,
    payment_date                        date           not null,
    discount_amount                     decimal(38, 2) not null,
    final_amount                        decimal(38, 2) not null,
    original_amount                     decimal(38, 2) not null,
    partner_business_registration_number varchar(100)  not null,
    partner_corp_name                   varchar(100)   not null
);

-- payment_date 컬럼에 대한 인덱스 생성 (조회 성능 향상)
create index idx_payment_date
    on payment_source (payment_date);
```

실습 Sample Data Setup

```
-- 10,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 10000);

-- 50,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 50000);

-- 100,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 100000);

-- 500,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 500000);

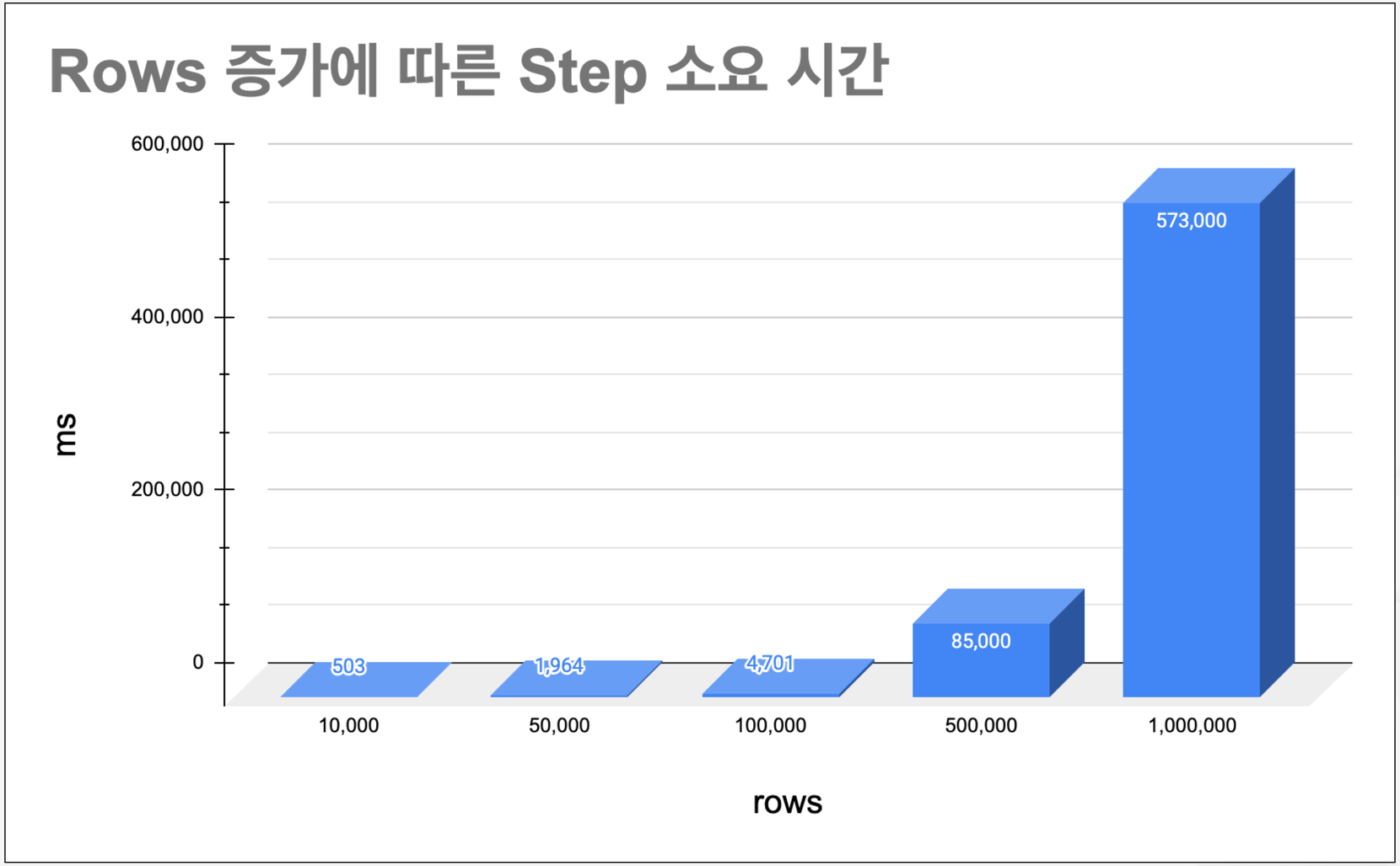
-- 1,000,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 1000000);

-- 5,000,000건 데이터 삽입
CALL sp_insert_payment_source_batch( p_row_count 5000000);
```

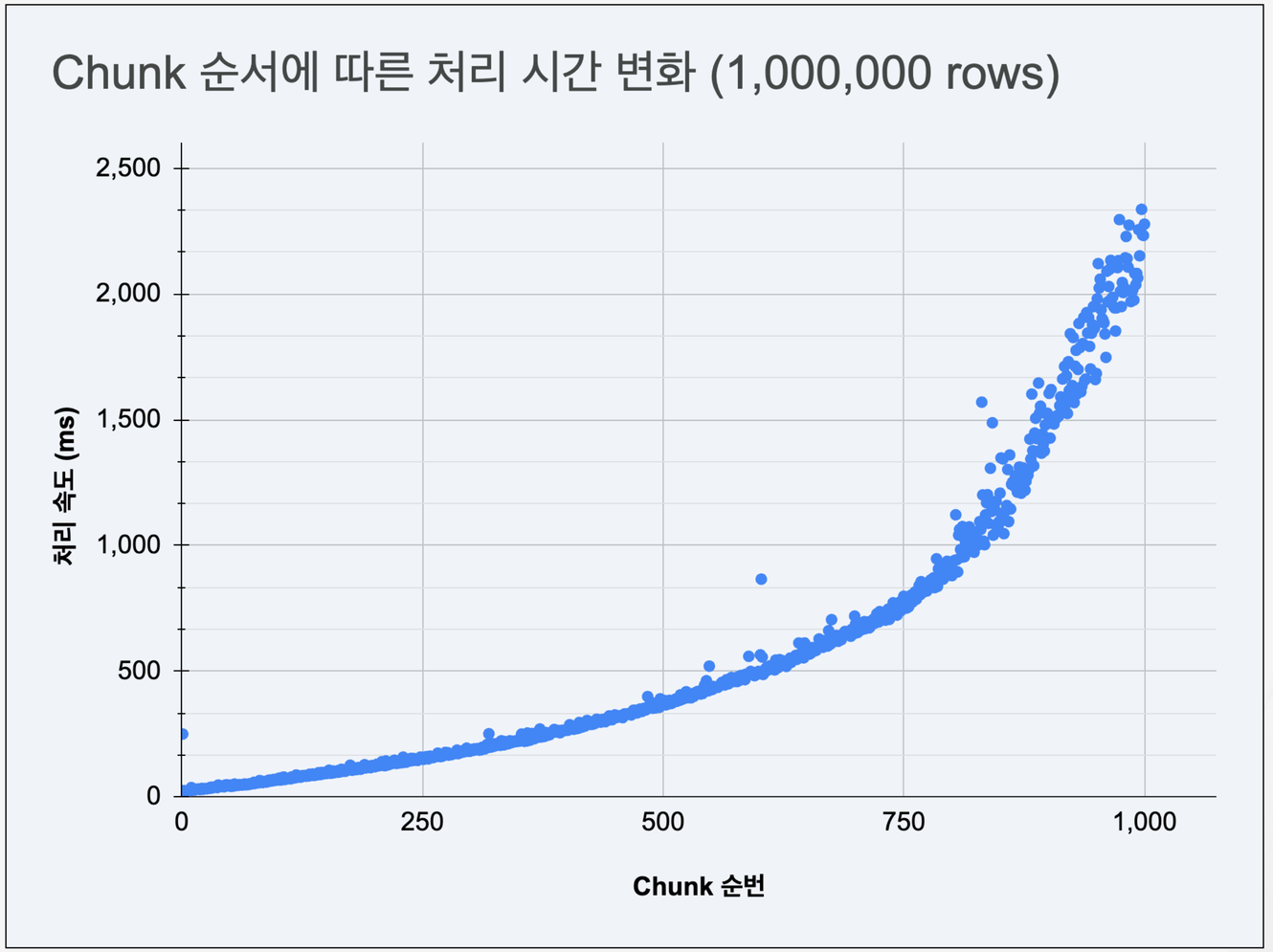
실습 Code

```
@Bean
public Step paymentReportStep(
        JpaPagingItemReader<PaymentSource> paymentReportReader
) {
    return new StepBuilder( name: "paymentReportStep", jobRepository)
            .<~>chunk( chunkSize: 1_000, transactionManager)
            // Step 소요 시간 측정
            .listener( listener: new StepDurationTrackerListener())
            .reader(paymentReportReader)
            .processor(paymentReportProcessor())
            .writer(paymentReportWriter())
            // Chunk 소요 시간 측장
            .listener( listener: new ChunkDurationTrackerListener())
            .build();
```

Limit-Offset Step 성능 측정



Limit-Offset Chunk 성능 측정



Limit-Offset 느려지는 원인

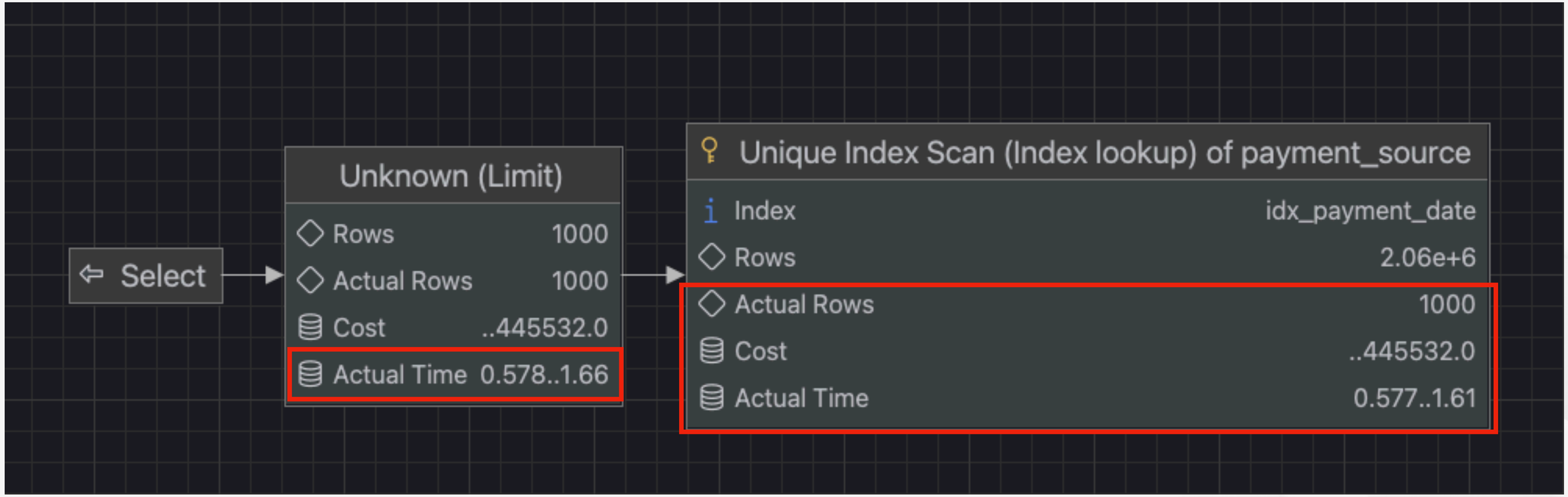


1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

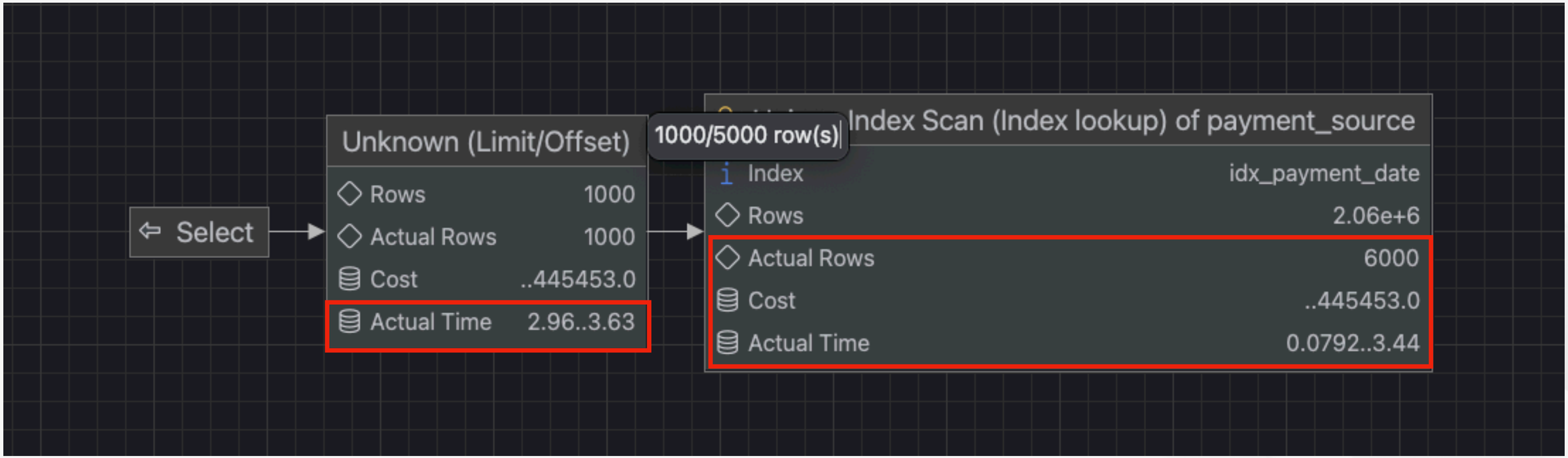
payment_date count



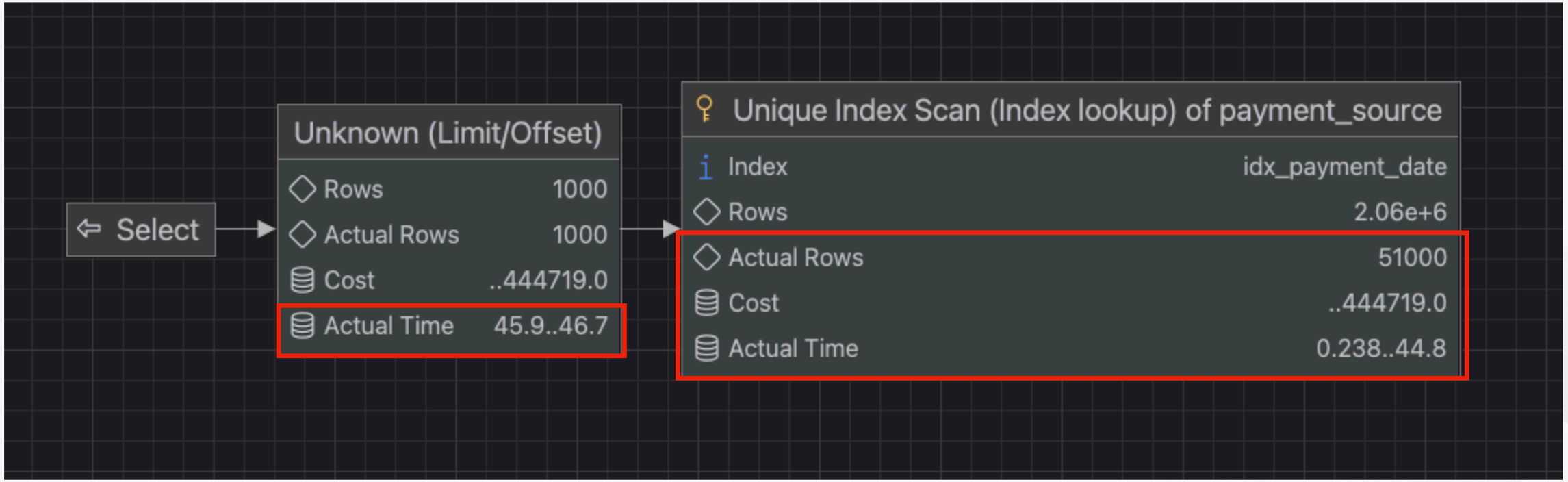
explain limit 1,000 offset 0



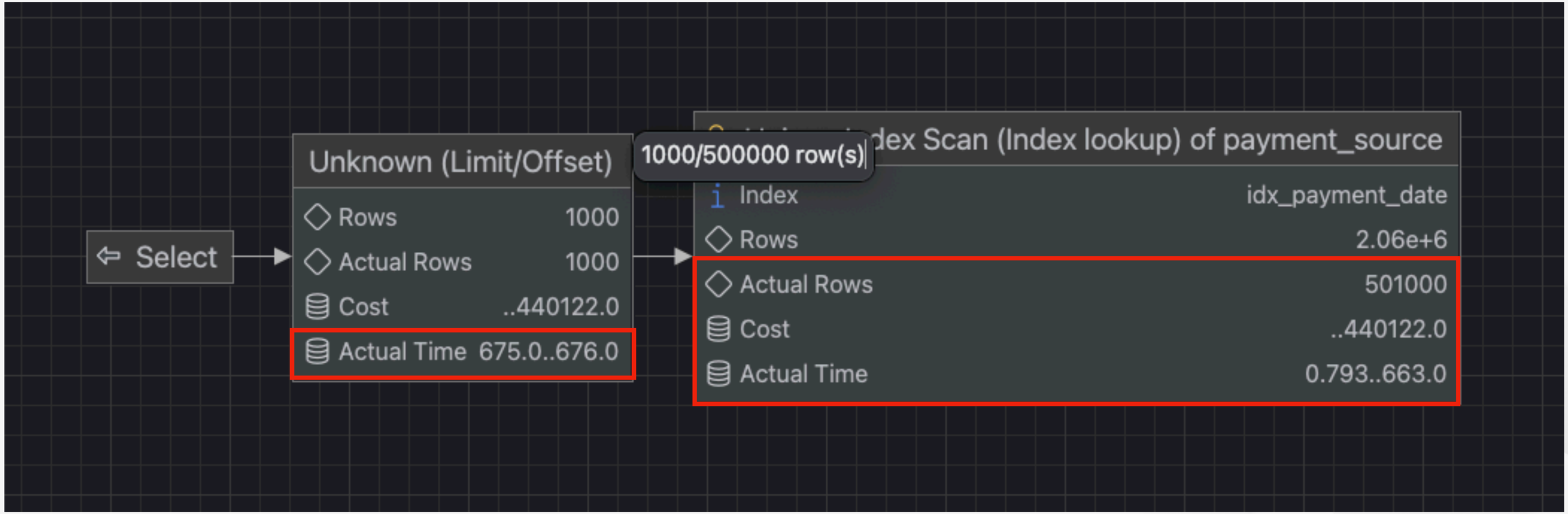
explain limit 1,000 offset 5,000



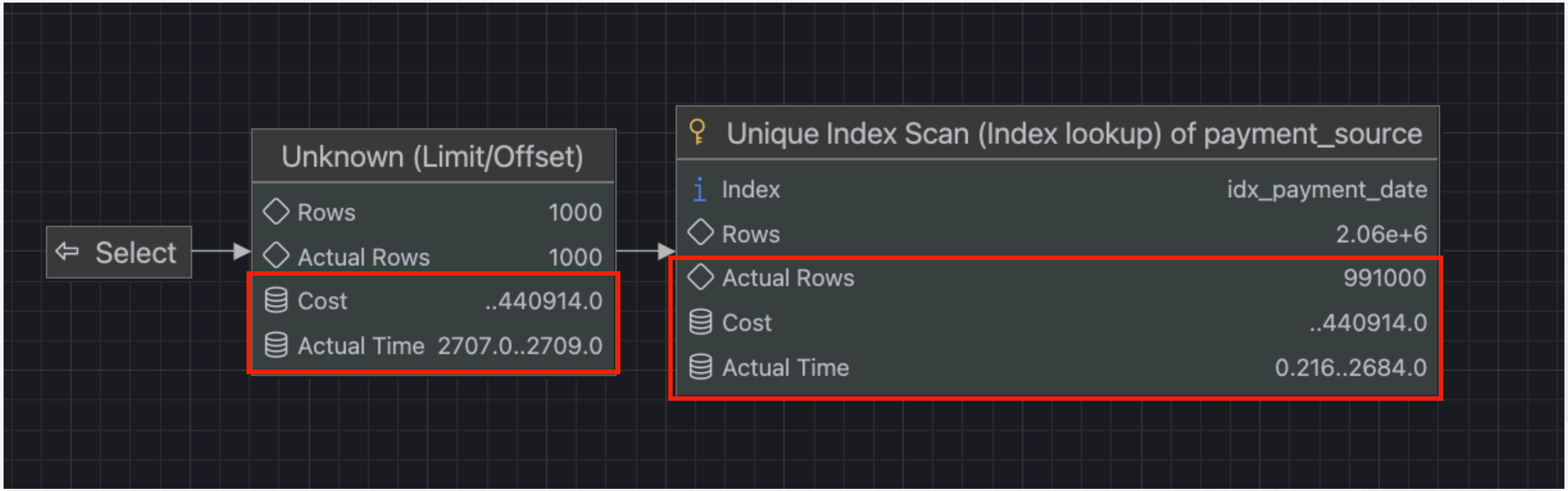
explain limit 1,000 offset 50,000



explain limit 1,000 offset 500,000



explain limit 1,000 offset 990,000



Limit Offset 느려지는 원인 - 데이터

2025-05-02 Count	Other Dates Count
1,000,000 rows	11,660,000 rows

Limit Offset 느려지는 원인 - offset 0

```
spring_batch> select id,
                     discount_amount,
                     final_amount,
                     original_amount,
                     partner_business_registration_number,
                     partner_corp_name,
                     payment_date
              from payment_source
              where payment_date = '2025-05-02'
              limit 1000 offset 0
[2025-07-13 13:00:53] 1,000 rows retrieved starting from 1 in 192 ms
 (execution: 15 ms, fetching: 177 ms)
```

Limit Offset 느려지는 원인 - offset 5,000

```
spring_batch> select id,
                     discount_amount,
                     final_amount,
                     original_amount,
                     partner_business_registration_number,
                     partner_corp_name,
                     payment_date
              from payment_source
              where payment_date = '2025-05-02'
              limit 10000 offset 50000
[2025-07-13 13:03:51] 10,000 rows retrieved starting from 1 in 399 ms
 (execution: 74 ms, fetching: 325 ms)
```

Limit Offset 느려지는 원인 - offset 50,000

```
spring_batch> select id,
                     discount_amount,
                     final_amount,
                     original_amount,
                     partner_business_registration_number,
                     partner_corp_name,
                     payment_date
              from payment_source
              where payment_date = '2025-05-02'
              limit 10000 offset 500000
[2025-07-13 13:06:03] 10,000 rows retrieved starting from 1 in 1 s 546
 ms (execution: 1 s 146 ms, fetching: 400 ms)
```

Limit Offset 느려지는 원인 - offset 990,000

```
spring_batch> select id,
                     discount_amount,
                     final_amount,
                     original_amount,
                     partner_business_registration_number,
                     partner_corp_name,
                     payment_date
              from payment_source
              where payment_date = '2025-05-02'
              limit 10000 offset 990000
[2025-07-13 13:09:28] 10,000 rows retrieved starting from 1 in 2 s 538
 ms (execution: 2 s 343 ms, fetching: 195 ms)
```

Limit Offset 느려지는 원인 - 불필요한 데이터 접근

Id	payment_date
13,380,001	2025-05-02
13,380,002	2025-05-02
….	2025-05-02
13,390,000	2025-05-02

limit 1,000 offset 0 데이터 조회

Limit Offset 느려지는 원인 - 불필요한 데이터 접근

Id	payment_date
13,380,001	2025-05-02
13,380,002	2025-05-02
….	2025-05-02
13,390,000	**2025-05-02**
13,381,001	2025-05-02
13,381,002	2025-05-02
…	2025-05-02
13,391,000	2025-05-02

실제 불필요 데이터

limit 1,000 offset 0 데이터 조회

실제 필요 데이터

limit 1,000 offset 1,000 데이터 조회

Limit Offset 느려지는 원인 - 불필요한 데이터 접근

Id	payment_date
14,369,001	2025-05-02
14,369,002	2025-05-02
….	2025-05-02
14,379,000	**2025-05-02**
14,370,001	2025-05-02
14,370,001	2025-05-02
…	2025-05-02
14,380,000	2025-05-02

실제 불필요 데이터

limit 1,000 offset 989,000 데이터 조회

실제 필요 데이터

limit 1,000 offset 999,000 데이터 조회

Limit Offset 속도 개선 방법

No-Offset 방식

- WHERE lastKey>... + LIMIT (키셋)
- 인덱스 Seek 기반, 일관된 성능
- 애플리케이션에서 상태(마지막 키) 관리

Cursor 방식

- DB에 커서 열고 한 행씩 순차적으로 읽기
- JDBC 서버-사이드 커서(fetchSize)
- 스트리밍 읽기, 메모리 최소화
- 단일 스레드 & 커넥션 장기 점유

No-Offset 방식의 데이터 접근 방식

Id	payment_date
14,369,001	2025-05-02
14,369,002	2025-05-02
….	2025-05-02
14,379,000	**2025-05-02**
14,370,001	2025-05-02
14,370,001	2025-05-02
…	2025-05-02
14,380,000	2025-05-02

실제 불필요 데이터

실제 필요 데이터

No-Offset 방식의 데이터 접근 방식

Id	payment_date
14,370,001	2025-05-02
14,370,001	2025-05-02
…	2025-05-02
14,380,000	2025-05-02

실제 필요 데이터에 바로 접근

No-Offset 방식 기존 방식

```sql
select id,
       discount_amount,
       final_amount,
       original_amount,
       partner_business_registration_number,
       partner_corp_name,
       payment_date
from payment_source
where payment_date = '2025-05-02'
limit 1000 offset 1000;
```

No-Offset 방식 Offset 제거

```
select id,
       discount_amount,
       final_amount,
       original_amount,
       partner_business_registration_number,
       partner_corp_name,
       payment_date
from payment_source
where payment_date = '2025-05-02'
limit 1000 offset 1000;
```

No-Offset 방식 Primary Key 기반 조회

```
select ps1_0.id,
       ps1_0.discount_amount,
       ps1_0.final_amount,
       ps1_0.original_amount,
       ps1_0.partner_business_registration_number,
       ps1_0.partner_corp_name,
       ps1_0.payment_date
from payment_source ps1_0
where ps1_0.id < 16050001
  and ps1_0.payment_date = '2025-05-02'
order by ps1_0.id desc
limit 1000;
```

No-Offset 방식 Primary Key 기반 조회

```
select ps1_0.id,
       ps1_0.discount_amount,
       ps1_0.final_amount,
       ps1_0.original_amount,
       ps1_0.partner_business_registration_number,
       ps1_0.partner_corp_name,
       ps1_0.payment_date
from payment_source ps1_0
where ps1_0.id < 16049001
  and ps1_0.payment_date = '2025-05-02'
order by ps1_0.id desc
limit 1000;
```

Chunk Size 만큼 진행

No-Offset 방식 첫 Chunk

```
spring_batch> select ps1_0.id,
                     ps1_0.discount_amount,
                     ps1_0.final_amount,
                     ps1_0.original_amount,
                     ps1_0.partner_business_registration_number,
                     ps1_0.partner_corp_name,
                     ps1_0.payment_date
              from payment_source ps1_0
              where ps1_0.id < 16050001
                and ps1_0.payment_date = '2025-05-02'
              order by ps1_0.id desc
              limit 1000
[2025-07-20 20:05:35] 1,000 rows retrieved starting from 1 in 324 ms (execution: 8 ms, fetching: 316 ms)
```

No-Offset 방식 중간 Chunk

```
spring_batch> select ps1_0.id,
                     ps1_0.discount_amount,
                     ps1_0.final_amount,
                     ps1_0.original_amount,
                     ps1_0.partner_business_registration_number,
                     ps1_0.partner_corp_name,
                     ps1_0.payment_date
              from payment_source ps1_0
              where ps1_0.id < 15550001
                and ps1_0.payment_date = '2025-05-02'
              order by ps1_0.id desc
              limit 1000
[2025-07-20 20:07:07] 1,000 rows retrieved starting from 1 in 286 ms (execution: 8 ms, fetching: 278 ms)
```

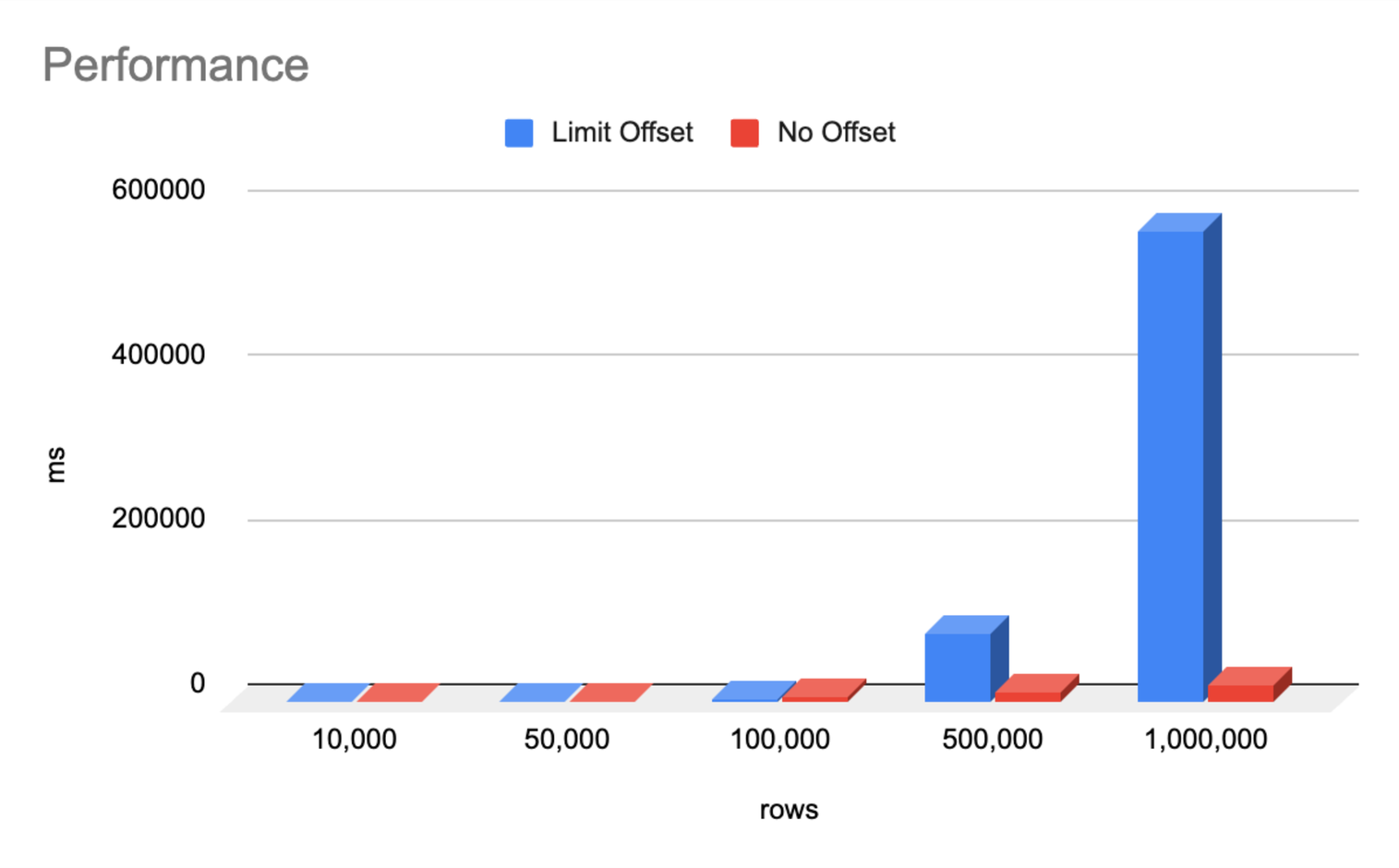
No-Offset 방식 후반 Chunk

```
spring_batch> select ps1_0.id,
                    ps1_0.discount_amount,
                    ps1_0.final_amount,
                    ps1_0.original_amount,
                    ps1_0.partner_business_registration_number,
                    ps1_0.partner_corp_name,
                    ps1_0.payment_date
             from payment_source ps1_0
             where ps1_0.id < 16050001
              and ps1_0.payment_date = '2025-05-02'
             order by ps1_0.id desc
             limit 1000
[2025-07-20 20:08:24] 1,000 rows retrieved starting from 1 in 402 ms (execution: 7 ms, fetching: 395 ms)
```

No-Offset 성능 비교

rows	Limit Offset	No Offset
10,000	503 ms	471 ms
50,000	1,964 ms	2,102 ms
100,000	4,701 ms	5,893 ms
500,000	85,000 ms	13,779 ms
1,000,000	**573,000 ms**	**21,953 ms**

No-Offset 성능 비교



No-Offset 동작 흐름

```
# 시작 지점 조회 쿼리
select
    id,
    discount_amount,
    final_amount,
    original_amount,
    partner_business_registration_number,
    partner_corp_name,
    payment_date
from payment_source
where payment_date = '2025-05-02' # 조회 조건
order by id desc
limit 1;
```

No-Offset 동작 흐름

```java
@Override
public void open( @NotNull ExecutionContext executionContext) throws ItemStreamException {
    this.entityManager = entityManagerFactory.createEntityManager();
    if (executionContext.containsKey( key: "firstId")) {
        // Job 재시작 시, 이전에 저장된 firstId를 가져와서 중단된 지점부터 다시 시작
        this.firstId = executionContext.getLong( key: "firstId");
    } else {
        // Job이 처음 시작될 때, 쿼리에서 가장 큰 ID 값을 가져와 firstId에 저장합니다.
        // 이 ID는 다음 페이지 조회의 시작점이 됩니다.
        TypedQuery<T> query = entityManager
                .createQuery( s: queryString,  aClass: this.targetType)
                .setMaxResults(1);
        parameterValues.forEach( action: query::setParameter);
        List<T> results = query.getResultList();

        if (results.isEmpty()) {
            // 조회 결과가 없으면 firstId를 0으로 설정
            this.firstId = 0L;
        } else {
            // 가장 큰 ID + 1을 시작점으로 설정하여 모든 데이터를 포함하도록 함
            this.firstId = idExtractor.apply( t: results.get(0)) + 1;
        }
    }
}
```

No-Offset 동작 흐름

```
# 첫 번째 Chunk 조회 쿼리
select id,
       discount_amount,
       final_amount,
       original_amount,
       partner_business_registration_number,
       partner_corp_name,
       payment_date
from payment_source
where payment_date = '2025-05-02'
  and id < 15051000 # 16,050,000(시작점 ID) + 1
#   and id < 15050000 # 16,050,000(시작점 ID) + 1
order by id desc
limit 1000; # chunk size
```

No-Offset 동작 흐름

```
@Override
public T read() {
    if (buffer.isEmpty() && !isEnd) {
        fillBuffer();
    }
    return buffer.poll();
}
```

No-Offset 동작 흐름

```
private void fillBuffer() {
    // 외부에서 받은 기본 쿼리(queryString)에 No-Offset 조건을 동적으로 추가합니다.
    // 사용자 쿼리에 WHERE 절이 있다는 전제 하에 AND로 연결합니다.
    String queryWithNoOffset = queryString.replace( target: "WHERE",  replacement: "WHERE id < :firstId AND");

    TypedQuery<T> query = entityManager
            .createQuery( s: queryWithNoOffset,  aClass: this.targetType) // 최종적으로 조립된 쿼리를 사용합니다.
            .setMaxResults(this.chunkSize);

    // 외부에서 주입된 파라미터 설정 (e.g., paymentDate)
    parameterValues.forEach( action: query::setParameter);
    // 내부 상태인 firstId 파라미터 설정
    query.setParameter( s: "firstId",  o: this.firstId);

    List<T> results = query.getResultList();

    if (results.isEmpty()) {
        // 조회 결과가 없으면 더 이상 읽을 데이터가 없음을 표시
        this.isEnd = true;
    } else {
        // 조회된 데이터를 버퍼에 추가하고, 다음 페이지 조회를 위해 firstId를 업데이트
        buffer.addAll( c: results);
        this.firstId = idExtractor.apply( t: results.get(results.size() - 1));
    }
}
```

No-Offset 동작 흐름

```java
/**
 * ItemStream의 상태를 업데이트합니다. Step 실행 중간에 주기적으로 호출됩니다.
 * 현재 페이지의 마지막 ID를 `firstId`로 ExecutionContext에 저장하여 Job 실패 시 복구할 수 있도록 합니다.
 *
 * @param executionContext Job 실행 컨텍스트
 * @throws ItemStreamException
 */
@Override
public void update( @NotNull ExecutionContext executionContext) throws ItemStreamException {
    executionContext.putLong("firstId", this.firstId);
}
```

스프링 배치 Reader 성능 개선

02

No-Offset 방식

- WHERE lastKey>... + LIMIT (키셋)
- 인덱스 Seek 기반, 일관된 성능
- 애플리케이션에서 상태(마지막 키) 관리

03

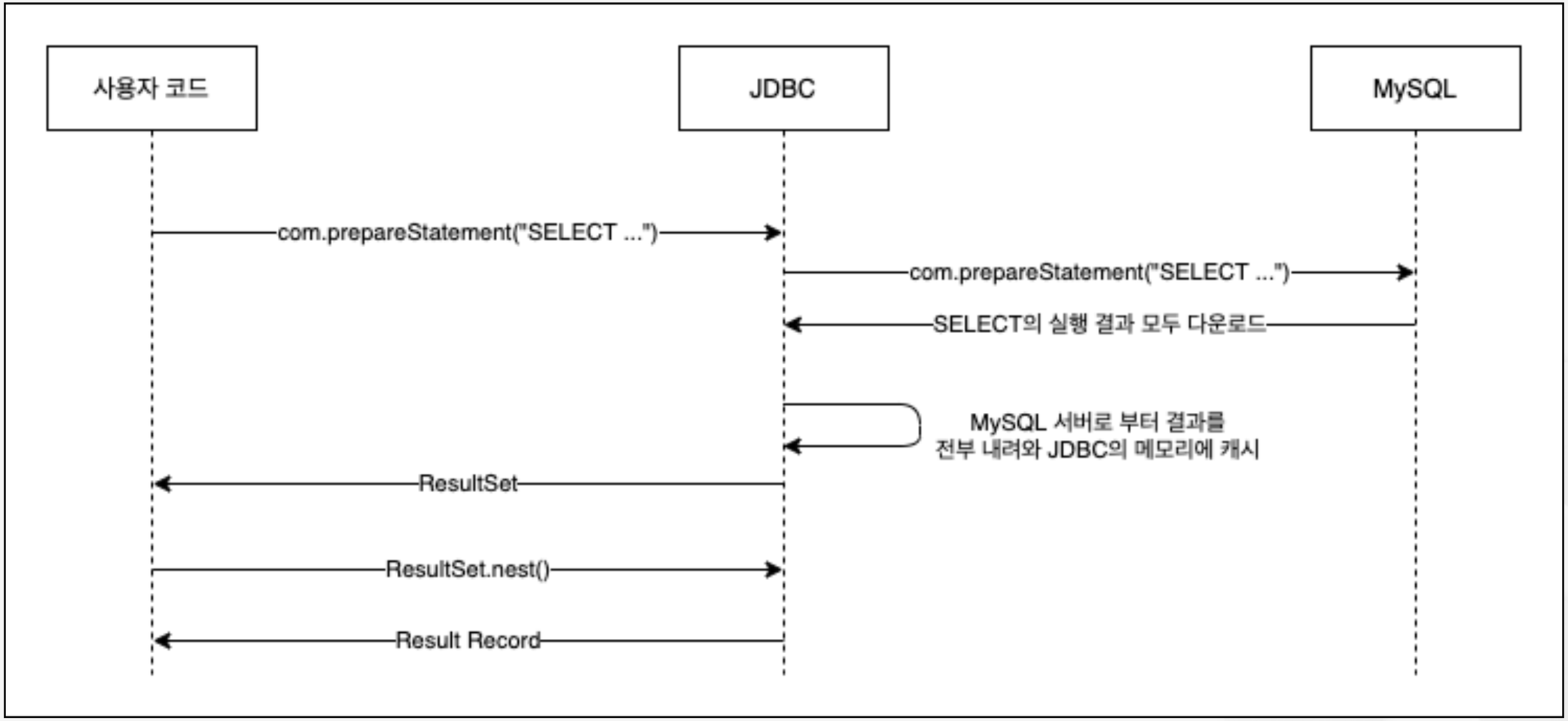
Cursor 방식

- DB에 커서 열고 한 행씩 순차적으로 읽기
- JDBC 서버-사이드 커서(fetchSize)
- 스트리밍 읽기, 메모리 최소화
- 단일 스레드 & 커넥션 장기 점유

Client-Side Cursor Streaming 이란?

- Client-Side Cursor Streaming 이란?
  - 클라이언트 사이드 커서는 **데이터베이스 쿼리의 모든 결과를 한 번에 애플리케이션의 메모리로 가져와서 처리하는 방식** 입니다.
  - MySQL JDBC 드라이버의 기본 동작 방식입니다.
- 동작 방식
  - 전체 데이터 다운로드: 애플리케이션이 `SELECT` 쿼리를 실행하면, 데이터베이스는 조회된 **모든 데이터를 클라이언트(애플리케이션)로 즉시 전송**합니다.
  - 메모리 캐시: JDBC 드라이버는 전송받은 모든 데이터를 **애플리케이션의 메모리(JVM Heap)에 저장**합니다.
  - 메모리에서 데이터 조회: 이후 애플리케이션이 `ResultSet.next()`를 호출하면, 데이터베이스와 추가 통신 없이 메모리에 저장된 데이터를 빠르게 읽어옵니다.
- 장점 및 단점
  - 장점: 일단 모든 데이터를 메모리에 가져온 후에는 데이터베이스 통신 없이 메모리에서 직접 읽으므로, **데이터 순회 속도가 매우 빠릅니다.**
  - 단점: 조회할 데이터가 많을 경우, 그 모든 데이터를 메모리에 저장해야 하므로 **메모리 사용량이 폭증**합니다. 이는 대용량 데이터를 처리할 때 **`OutOfMemoryError`를 유발하는 주된 원인이** 됩니다.

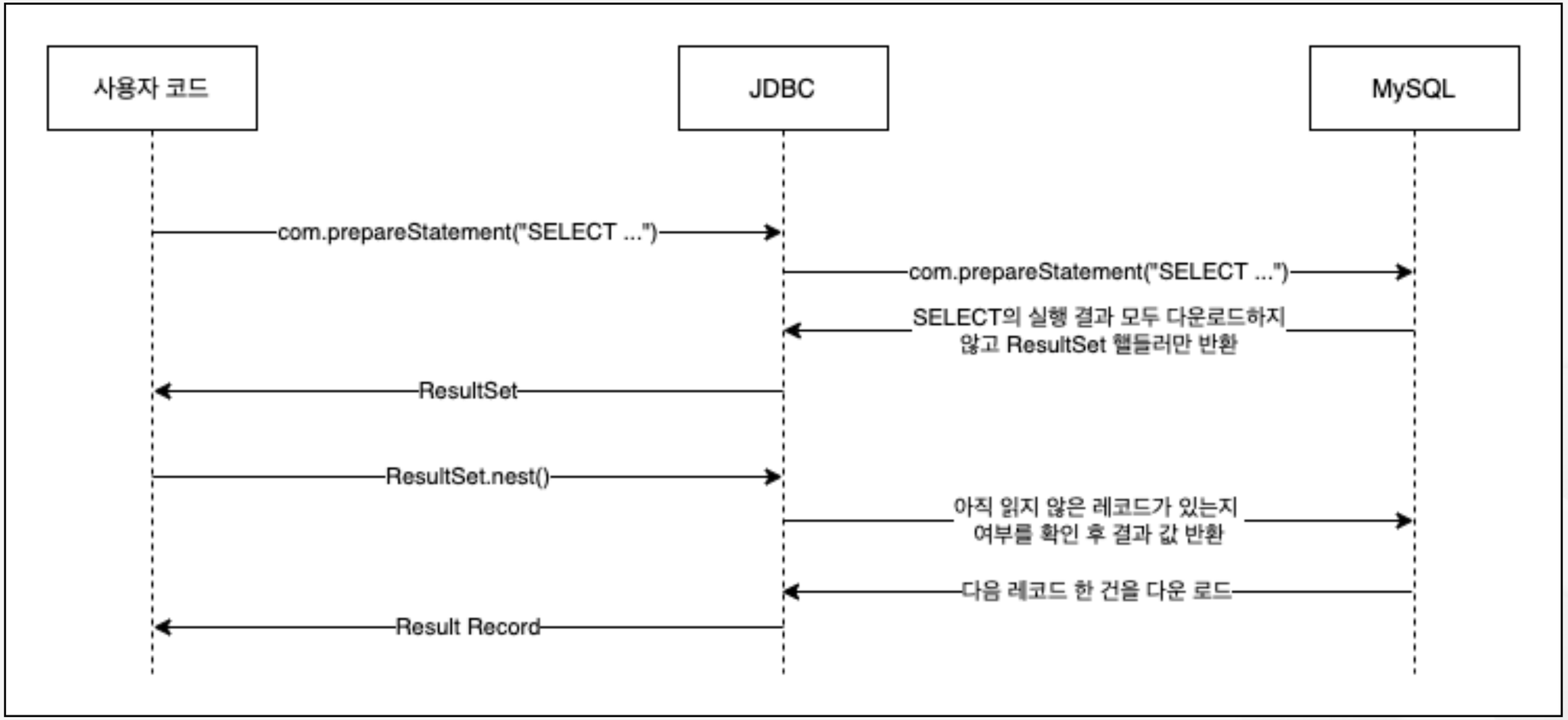
Client-Side Cursor Streaming Flow



Server-Side Cursor Streaming 이란?

- Server-Side Cursor Streaming 이란?
  - 서버 사이드 커서는 데이터베이스 쿼리의 결과를 서버에 유지한 채, 애플리케이션이 필요할 때마다 데이터를 조금씩 스트리밍으로 가져와 처리하는 방식입니다.
  - 대용량 데이터를 메모리 문제 없이 안정적으로 처리하기 위해 명시적인 설정이 필요한 방식입니다.
- 동작 방식
  - 쿼리 실행 및 커서 반환: 애플리케이션이 스트리밍 옵션과 함께 `SELECT` 쿼리를 실행하면, 데이터베이스는 결과 전체를 서버에 준비해두고 결과 집합을 가리키는 포인터(커서)만 즉시 반환합니다. 실제 데이터는 이때 전송되지 않습니다.
  - 데이터 요청 및 수신: 이후 애플리케이션이 `ResultSet.next()`를 호출할 때마다, 데이터베이스에 '다음 데이터 한 건'을 요청하는 네트워크 통신이 발생합니다.
  - 순차적 처리: 데이터베이스는 요청받은 데이터 한 건만 응답으로 보내주며, 이 과정이 모든 데이터를 다 읽을 때까지 반복됩니다.
- 장점 및 단점
  - 장점: 조회할 데이터가 아무리 많아도 애플리케이션의 메모리 사용량이 매우 낮고 일정하게 유지되므로 **`OutOfMemoryError`를 원천적으로 방지**할 수 있습니다.
  - 단점: 모든 데이터를 읽을 때까지 **데이터베이스 커넥션을 계속 유지**해야 하므로 DB에 부담을 줄 수 있으며, 데이터 건당 발생하는 네트워크 통신으로 인해 전체 처리 속도가 상대적으로 느릴 수 있습니다.

Server-Side Cursor Streaming Flow



Cursor 방식 정리

특성	클라이언트 사이드 커서 (버퍼링)	서버 사이드 커서 (스트리밍)
데이터 위치 (순회 중)	전체 결과셋이 클라이언트 애플리케이션 메모리(Java Heap)에 상주	대부분의 결과셋이 데이터베이스 서버에 상주
클라이언트 메모리 사용량	높음. 전체 결과셋 크기에 비례. OutOfMemoryError 발생 가능성 높음.	낮음. Fetch 크기에만 비례하여 일정하게 유지됨.
서버 리소스 사용량	낮음. 초기 데이터 전송 후 서버는 거의 무상태(stateless).	높음. 클라이언트가 순회하는 동안 커서 상태, 결과셋, 잠재적 락(lock)을 유지해야 함.
네트워크 트래픽 패턴	초기에 한 번의 대규모 데이터 전송.	"Chatty". 커서 수명 동안 다수의 작은 데이터 전송 발생.
연결 상태	데이터 전송 후 신속하게 연결 해제 가능.	순회가 끝날 때까지 연결이 열려 있고 상태를 유지해야 함(stateful).
이상적인 사용 사례	응답성이 중요한 중소 규모(수만 건 미만) 결과셋 처리.	대규모(수십만~수백만 건 이상) 결과셋을 처리하는 전형적인 배치 작업.
주요 단점	메모리 제약으로 인한 확장성 부족.	서버 부하 증가 및 연결 관리, 타임아웃의 복잡성.

Server-Side Cursor 으로 동작 시키기 위한 작업

- ***결과 집합***

  기본적으로 ResultSet은 메모리에 완전히 검색되어 저장됩니다. 대부분의 경우 이 방식이 가장 효율적인 운영 방식이며, MySQL 네트워크 프로토콜 설계 덕분에 구현이 더 쉽습니다. 행 수가 많거나 값이 큰 ResultSet을 사용 중이고 JVM에 필요한 메모리를 위한 힙 공간을 할당할 수 없는 경우, 드라이버에 결과를 한 번에 한 행씩 스트리밍하도록 설정할 수 있습니다.

  이 기능을 활성화하려면 `Statement`다음과 같은 방식으로 인스턴스를 생성하세요.

```
stmt = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                java.sql.ResultSet.CONCUR_READ_ONLY);
stmt.setFetchSize(Integer.MIN_VALUE);
```

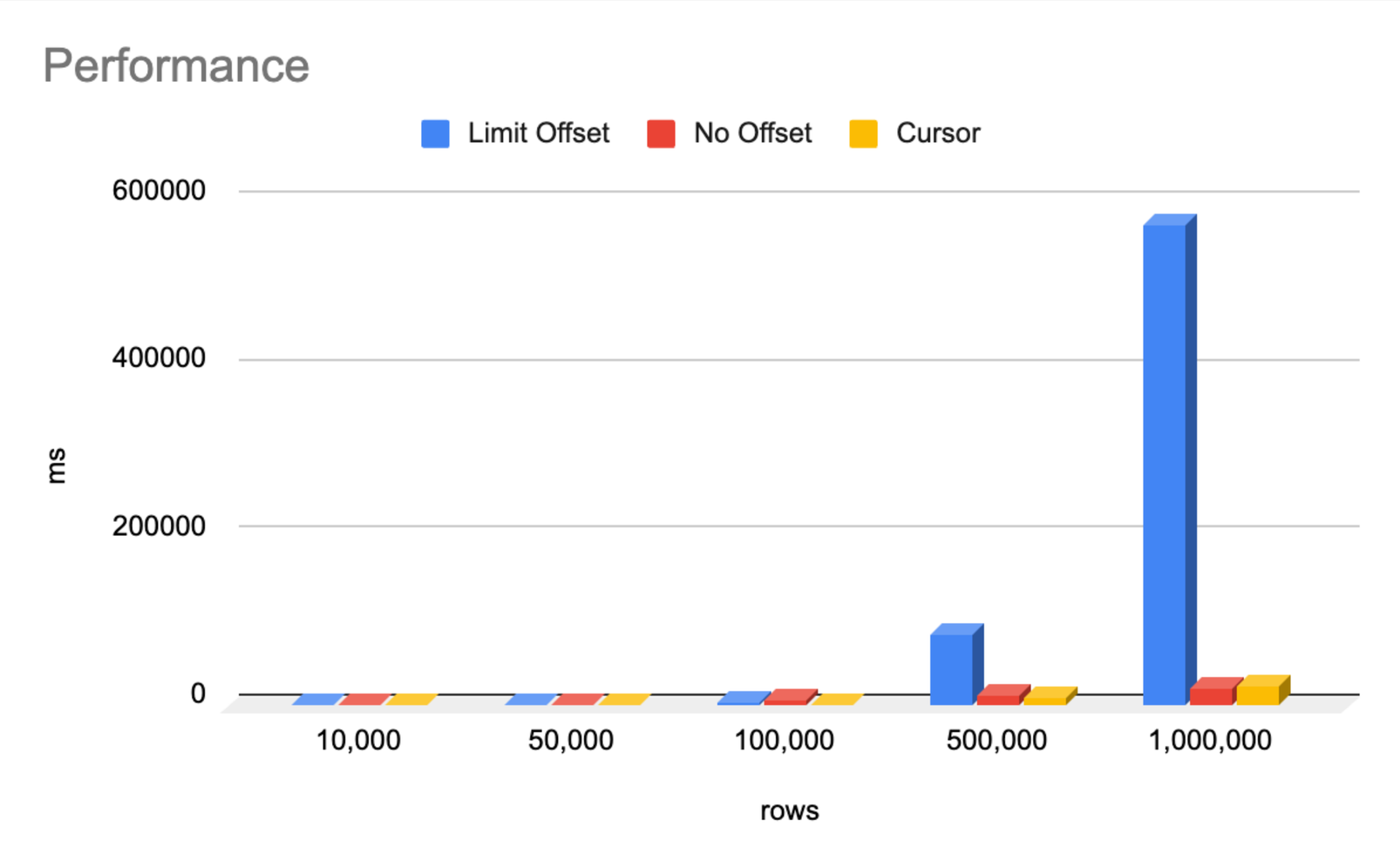
  전달 전용, 읽기 전용 결과 집합과 페치 크기(fetch size)의 조합은 `Integer.MIN_VALUE` 드라이버에 결과 집합을 행 단위로 스트리밍하라는 신호 역할을 합니다. 이후 해당 명령문으로 생성된 모든 결과 집합은 행 단위로 검색됩니다.

https://dev.mysql.com/doc/connector-j/en/connector-j-reference-implementation-notes.html

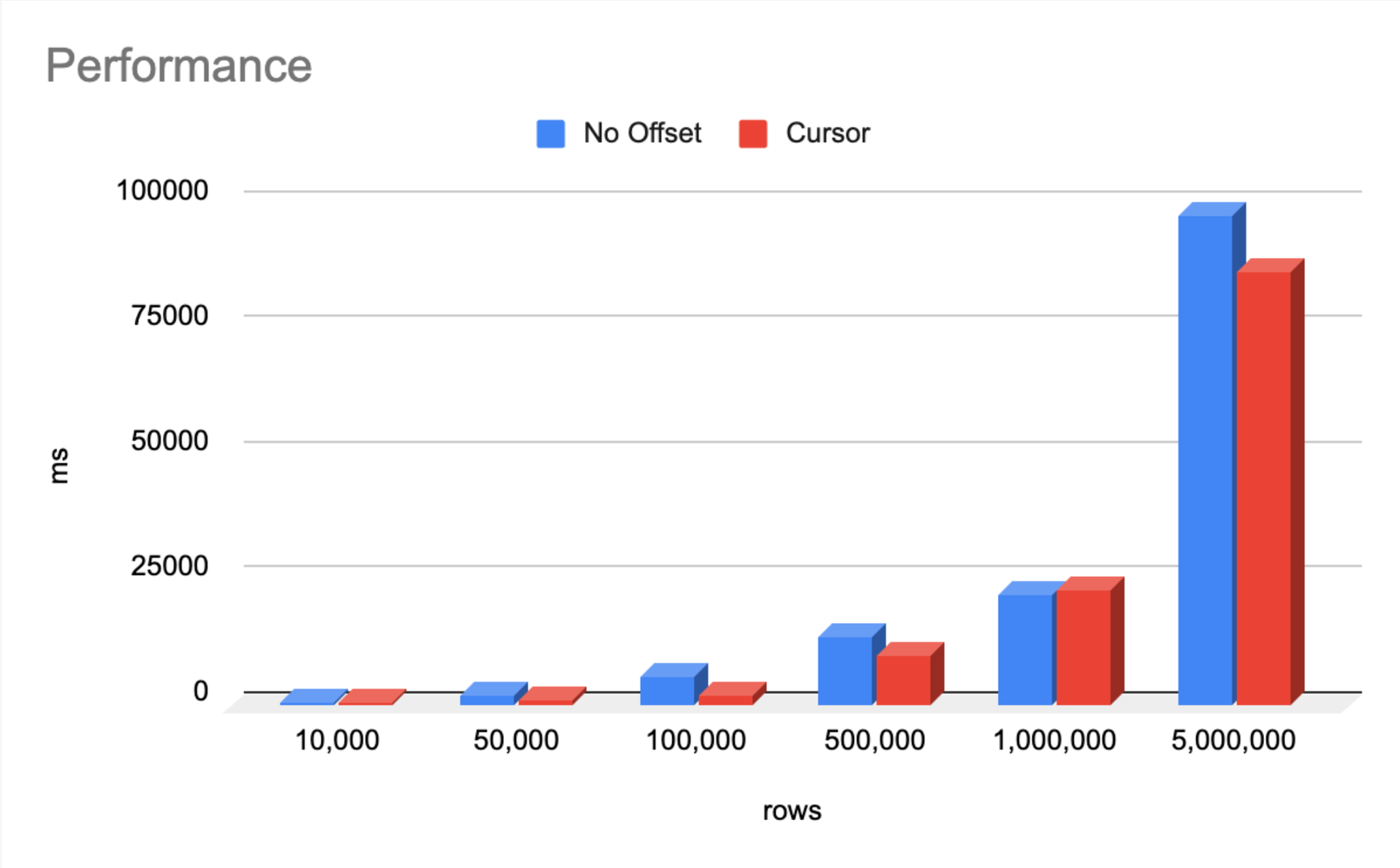
Server-Side Cursor 으로 동작 시키기 위한 작업

```
new JpaCursorItemReaderBuilder<PaymentSource>()
 .name( name: "cursorItemReader")
 .entityManagerFactory(entityManagerFactory)
 // JPQL 쿼리 작성: SQL이 아닌 JPQL(Java Persistence Query Language)을 사용
 .queryString( queryString: "SELECT ps FROM PaymentSource ps WHERE ps.paymentDate = :paymentDate")
 // 쿼리 파라미터 설정
 .parameterValues( parameterValues: Collections.singletonMap("paymentDate", paymentDate))
 // Hibernate Hint 설정: HINT_FETCH_SIZE
 // 이 옵션은 데이터베이스에서 데이터를 가져올 때 한 번에 한 행씩 가져오도록 DB에 힌트를 줍니다.  →| to complete   You, Today • Uncommitted changes
 // 서버 측 커서(Server-Side Cursor)를 사용할지 클라이언트 측 커서(Client-Side Cursor)를 사용할지 결정하는 중요한 역할을 합니다.
 // Integer.MIN_VALUE: 이 값을 설정하면 JDBC 드라이버에게 결과를 스트리밍 방식으로 받아오도록 요청합니다.
 // 즉, 데이터베이스 서버에 커서를 유지한 채로, 애플리케이션에서는 next()를 호출할 때마다 한 건씩 데이터를 가져옵니다.
 // 이를 통해 대용량의 데이터셋을 처리하더라도 애플리케이션의 메모리 사용량을 최소화할 수 있어 OOM(OutOfMemory) 에러를 방지합니다.
 // 만약 이 옵션이 없다면, JPA 구현체(Hibernate)는 기본적으로 조회된 모든 결과를 애플리케이션 메모리로 가져와서(Client-Side Cursor) 처리하려고 시도할 수 있습니다.
 .hintValues( hintValues: Collections.singletonMap(HibernateHints.HINT_FETCH_SIZE, Integer.MIN_VALUE))
 .build();
```

Cursor 성능 비교


Performance
Limit Offset
No Offset
Cursor
600000
400000
200000
0
ms
10,000
50,000
100,000
500,000
1,000,000
rows

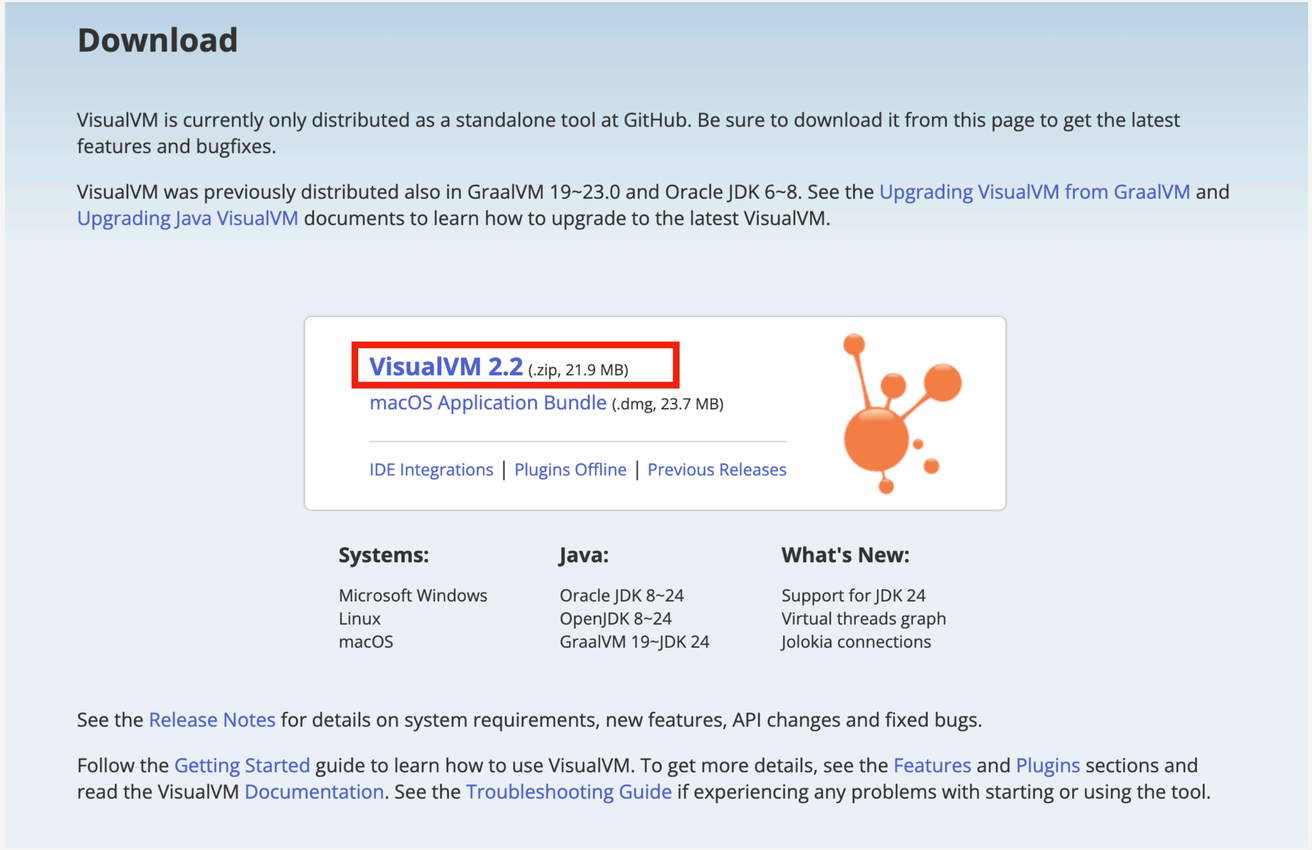
Cursor 성능 비교


Performance
No Offset
Cursor
100000
75000
50000
25000
0
ms
10,000
50,000
100,000
500,000
1,000,000
5,000,000
rows

Cursor 성능 비교

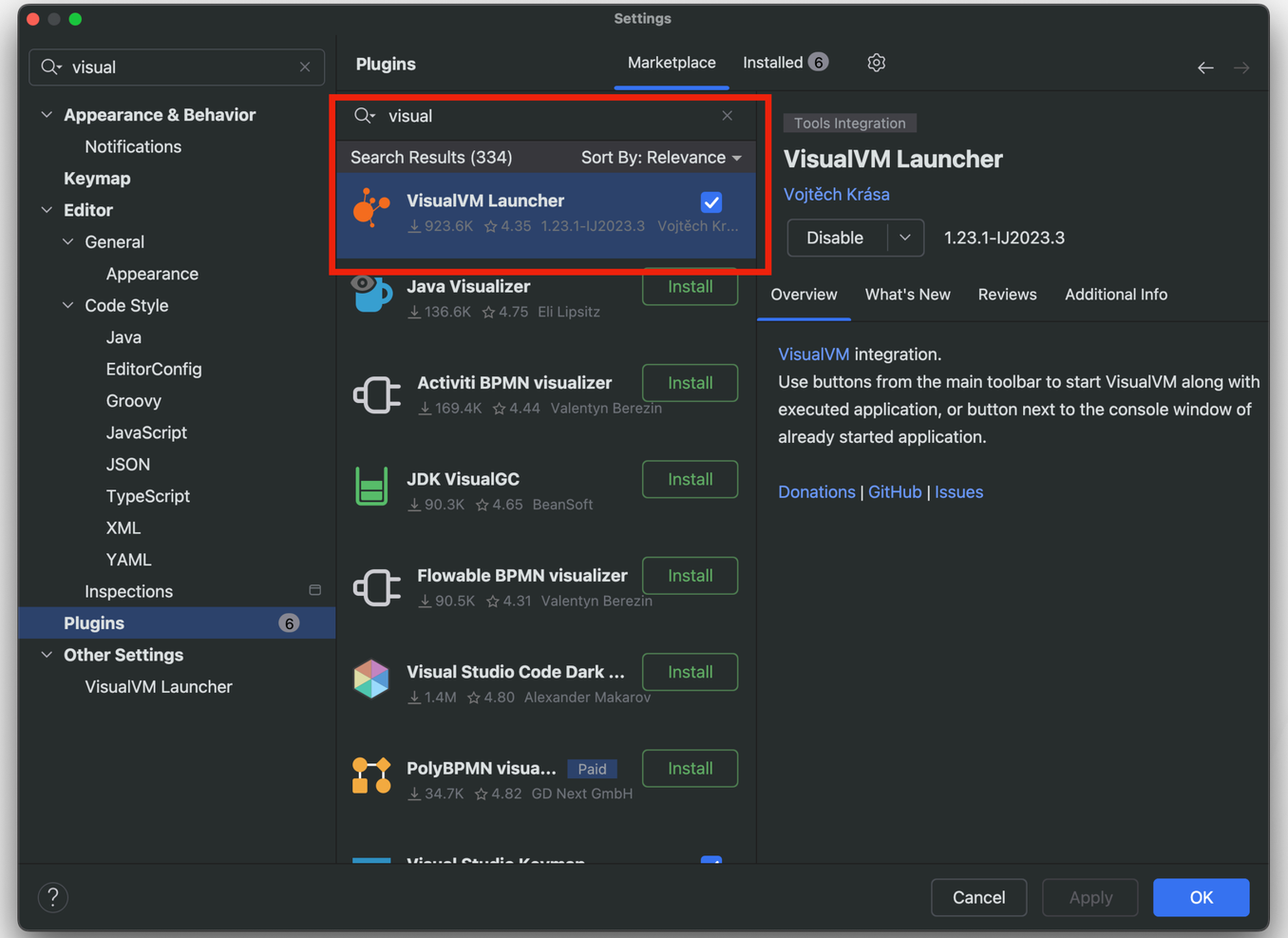
rows	No Offset	Cursor (Client)
10,000	471 ms	472 ms
50,000	2,102 ms	1,239 ms
100,000	5,893 ms	2,108 ms
500,000	13,779 ms	9,840 ms
1,000,000	21,953 ms	23,104 ms
5,000,000	**97,587 ms**	**86,381 ms**

VisualVM Launcher Zip Download

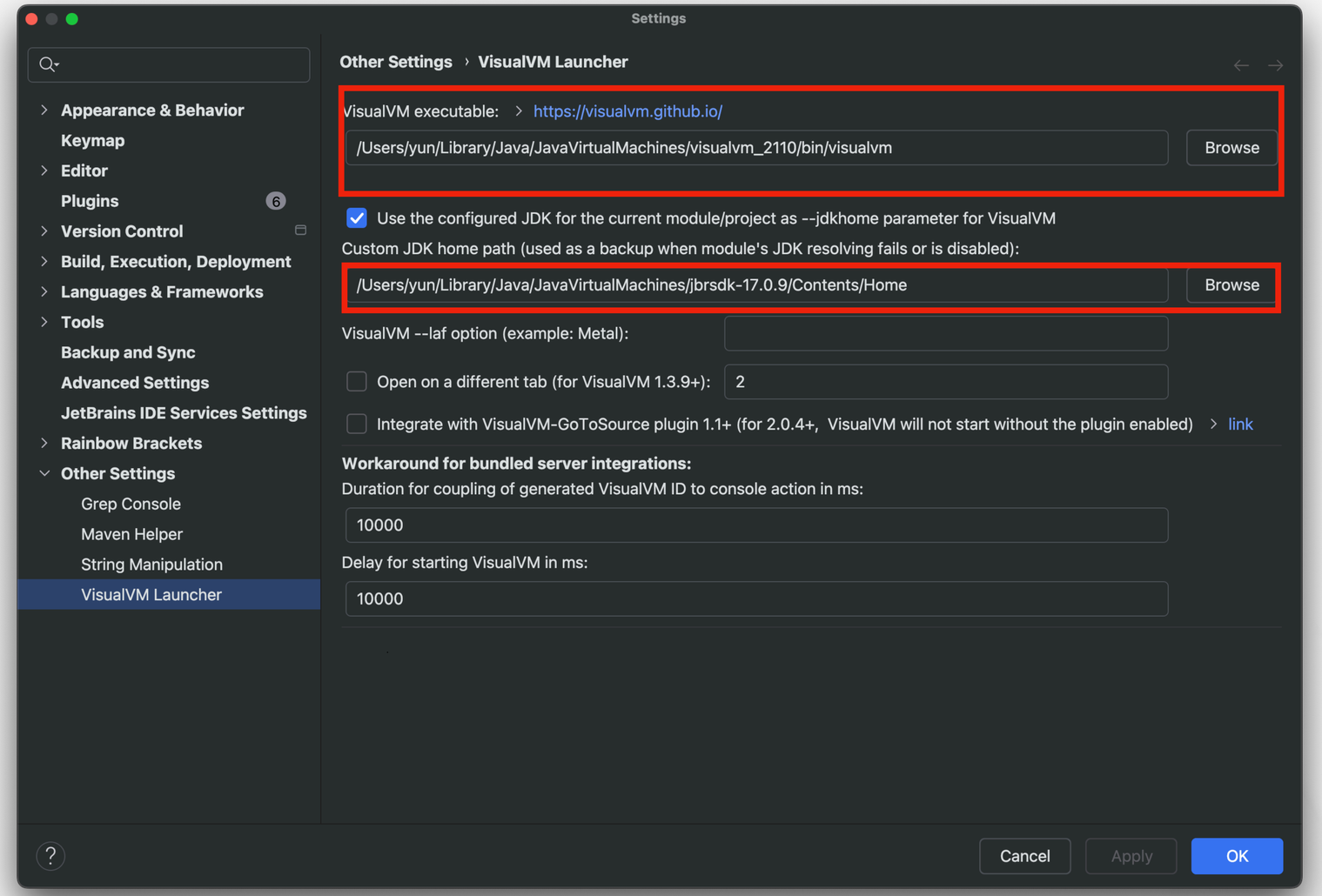


https://visualvm.github.io/download.html

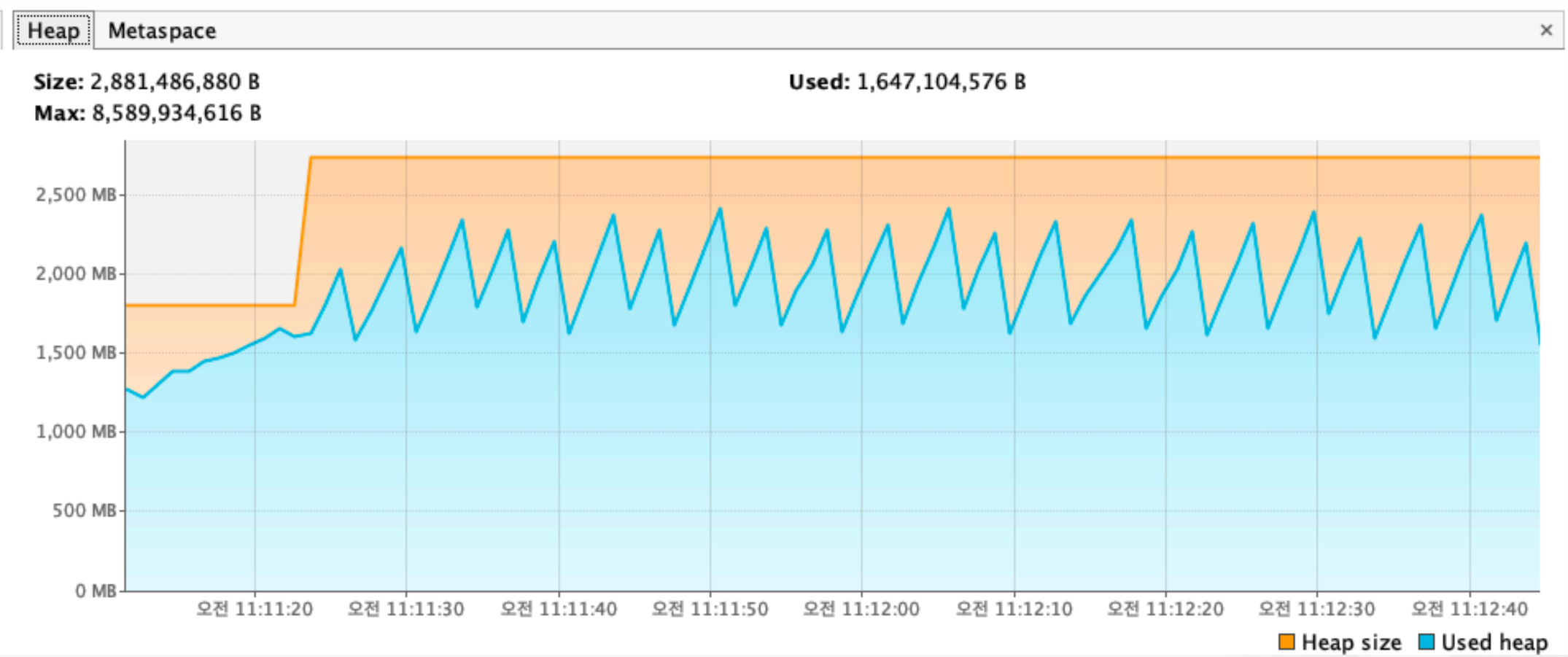
VisualVM Launcher Plugin Install



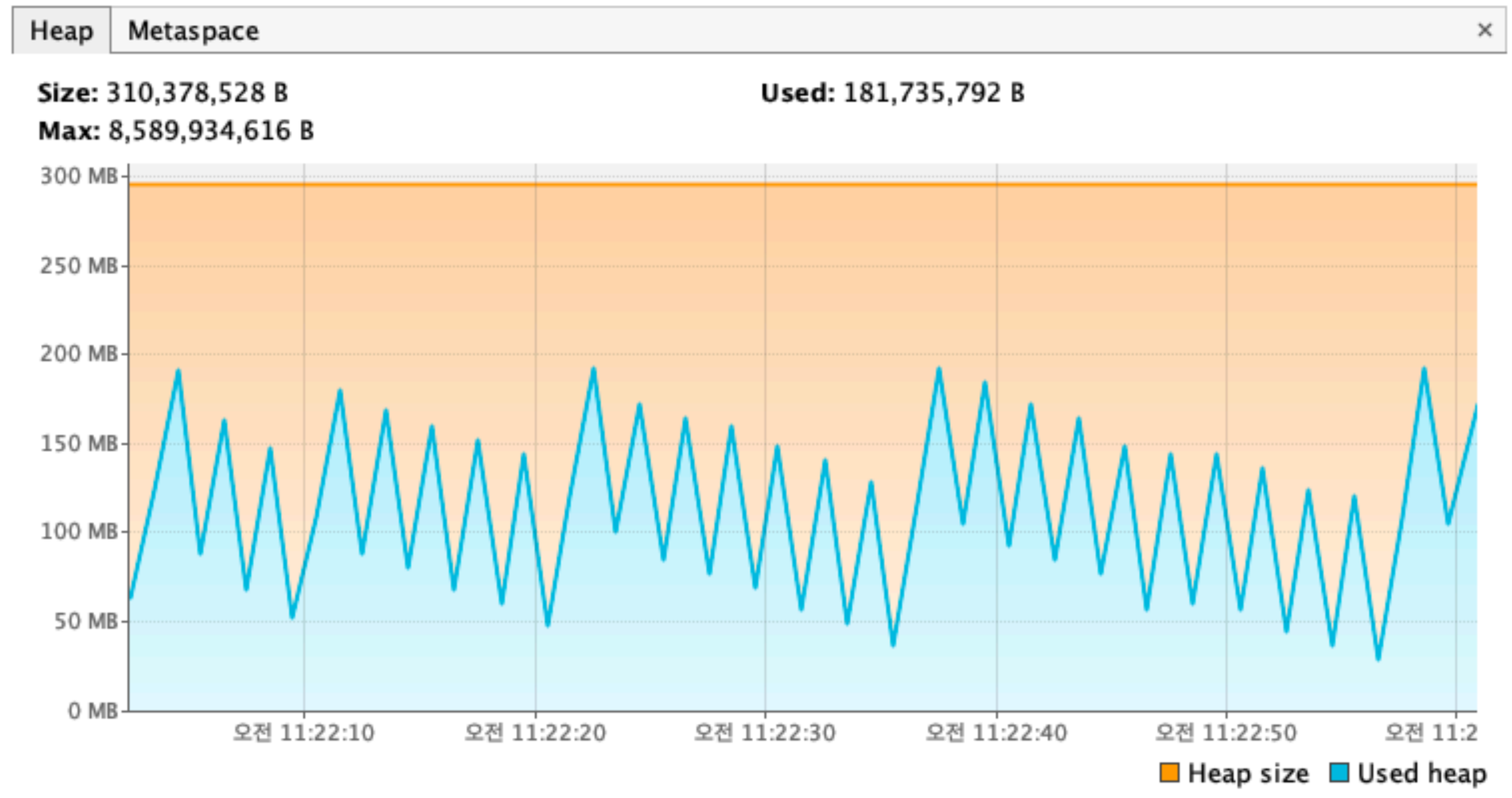
VisualVM Launcher Plugin Setting



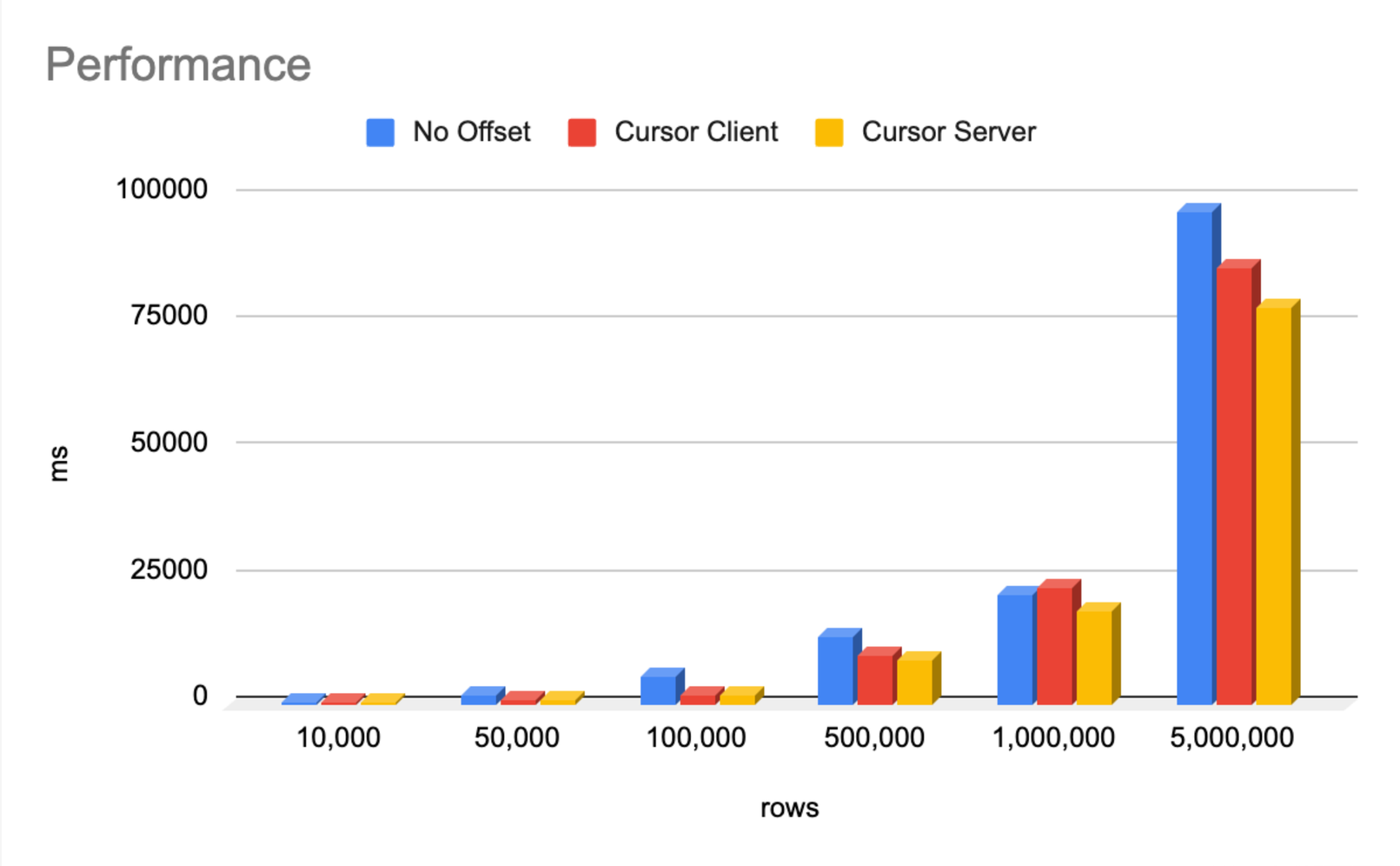
Client-Side Cursor rows 5,000,000



Server-Side Cursor rows 5,000,000



Cursor Client, Server 성능 비교



Cursor Client, Server 성능 비교

rows	No Offset	Client Side Cursor	Server Side Cursor
10,000	471 ms	472 ms	476 ms
50,000	2,102 ms	1,239 ms	1,213 ms
100,000	5,893 ms	2,108 ms	2,111 ms
500,000	13,779 ms	9,840 ms	9,016 ms
1,000,000	21,953 ms	23,104 ms	18,814 ms
5,000,000	**97,587 ms**	**86,381 ms**	**78,288 ms**

Part 2 Spring Batch 성능 최적화 기법

01

스프링 배치 성능 개선 Reader

- Limit offset 방식
- Zero offset 방식
- Cursor 방식

02

스프링 배치 성능 개선 Processor & Writer

- Processor 성능 개선
- Writer 성능 개선
- 병목 원인 파악 및 개선

03

스프링 배치 성능 개선 Partition Step

- 병렬 Partition Step 처리
- MasterStep, Slave Step 수행 구조

스프링 배치 성능 개선

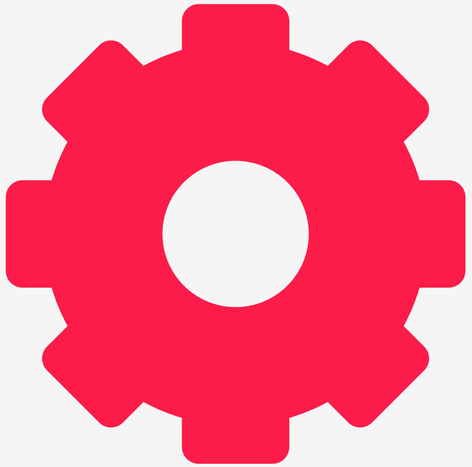
Processor 성능 개선

- I/O 병목 최소화를 위해 Processor 로직을 Writer로 이전하여 일괄 처리
- Writer 내에서 외부 API 호출 등은 RX Java 등으로 병렬화하여 처리 속도 향상

Writer 성능 개선개선

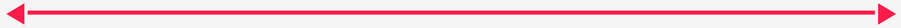
- 데이터를 그룹화하여 WHERE IN 절로 한번에 업데이트 (DB I/O 감소)
- 불필요한 데이터베이스 접근을 제거하여 처리 속도 향상

회원 등급 Batch Flow



회원 등급 업데이트
Batch

API /users/{userId}/grade
회원 주문 등급 Response

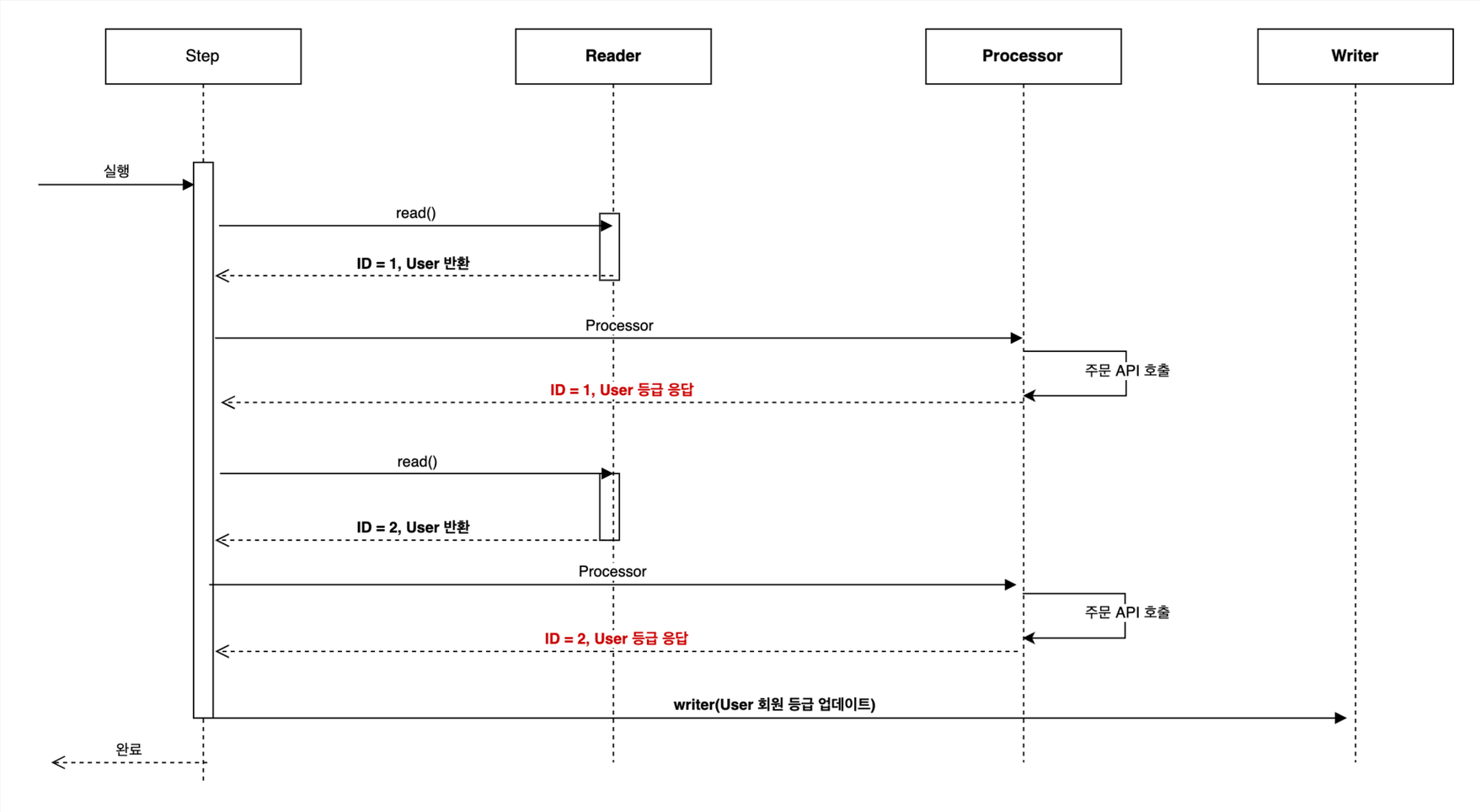


```
public enum Grade {
    INIT("초기 등급"),
    BASIC("기본 등급"),
    PREMIUM("프리미엄 등급"),
    VIP("VIP 등급");
}
```

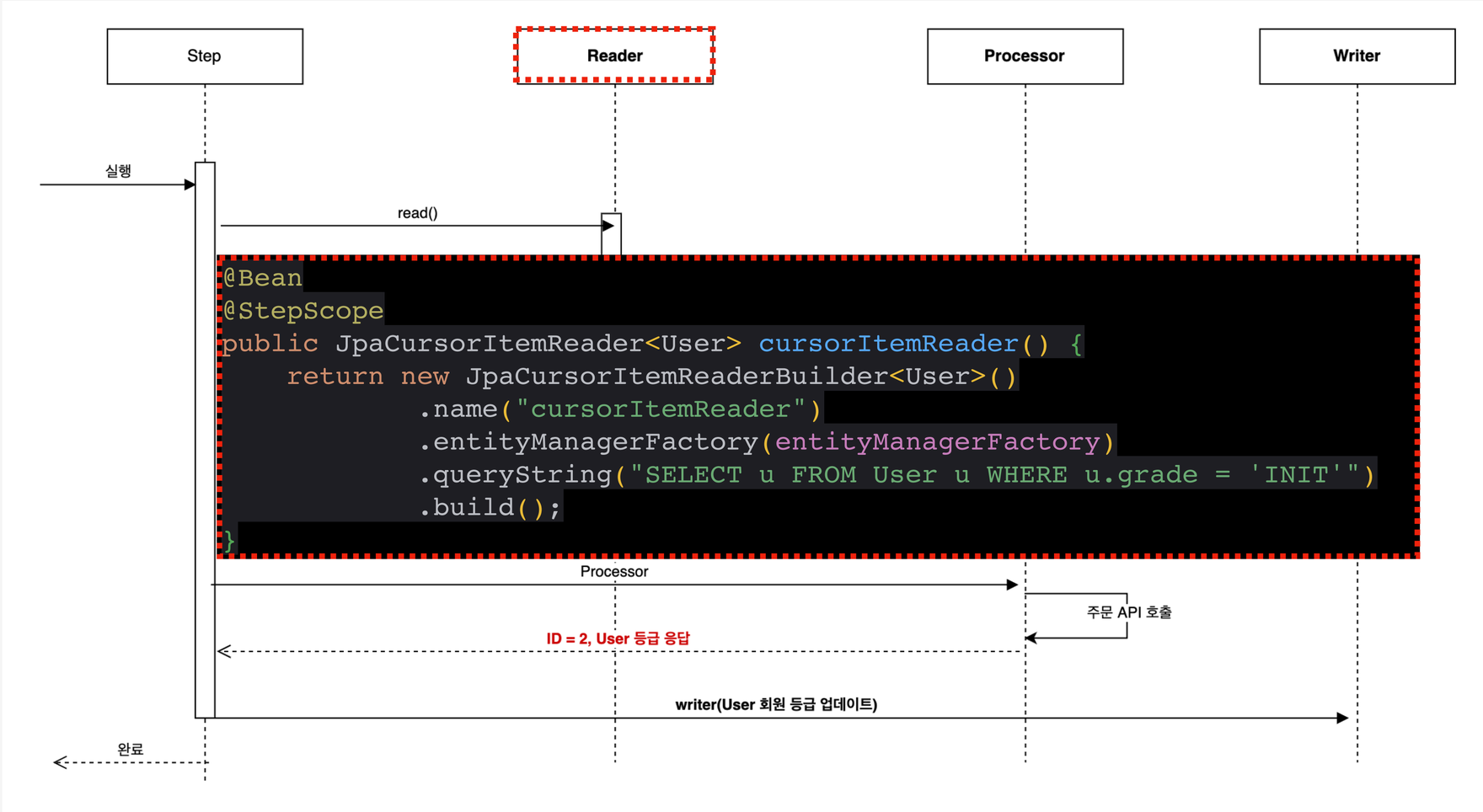


주문 API

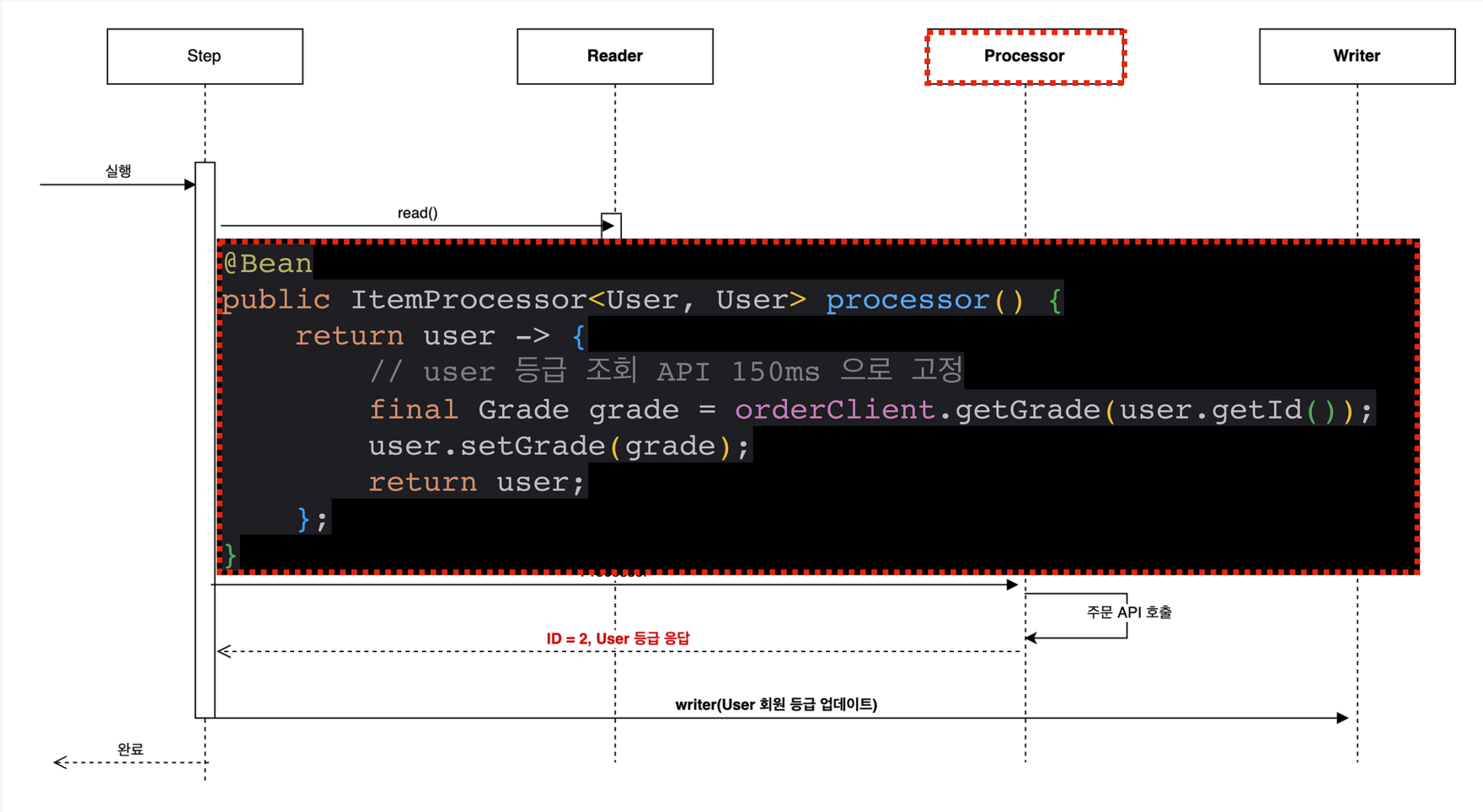
회원 등급 Batch Flow



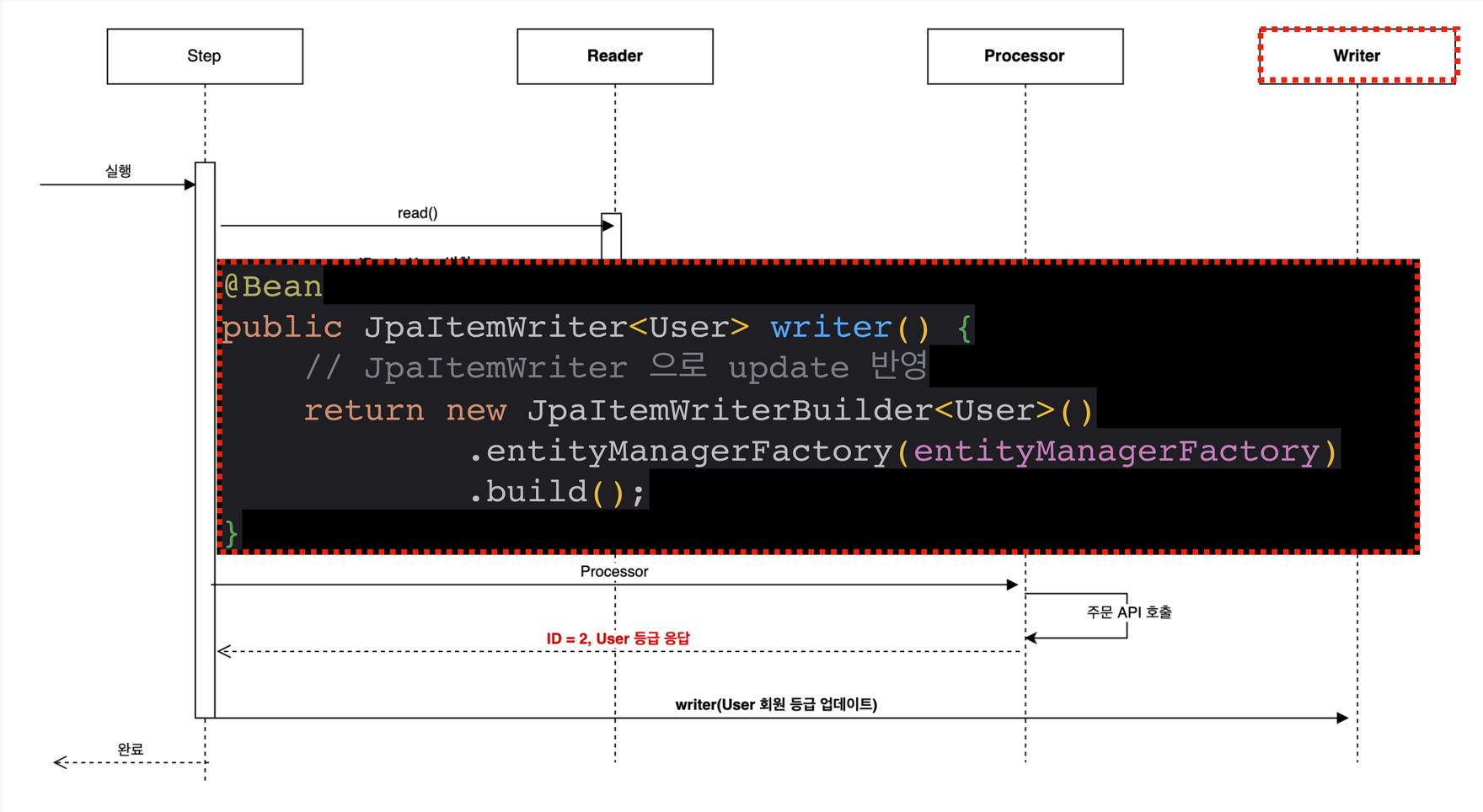
회원 등급 Batch Flow Reader



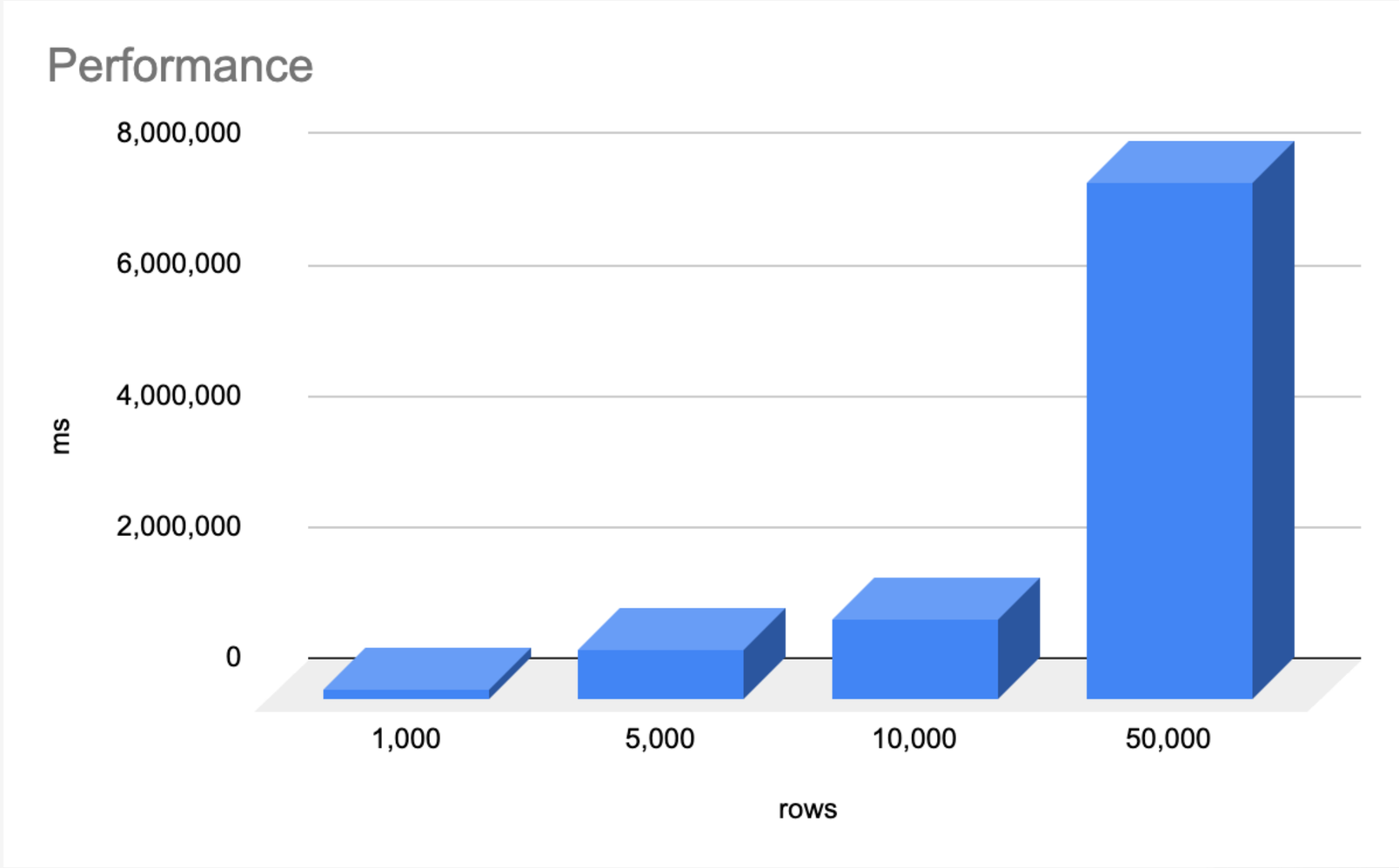
회원 등급 Batch Flow Processor



회원 등급 Batch Flow Writer



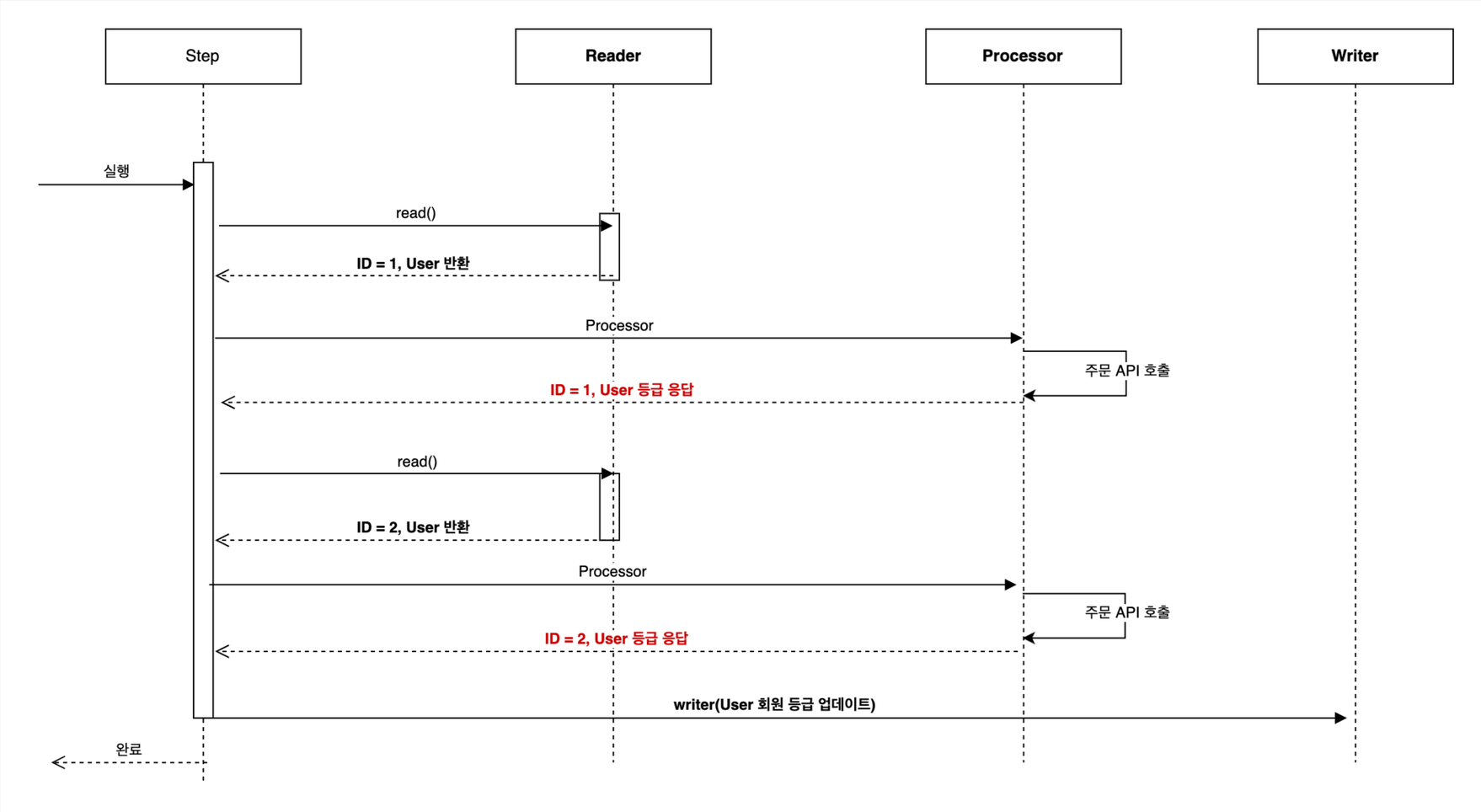
회원 등급 Batch 성능 측정



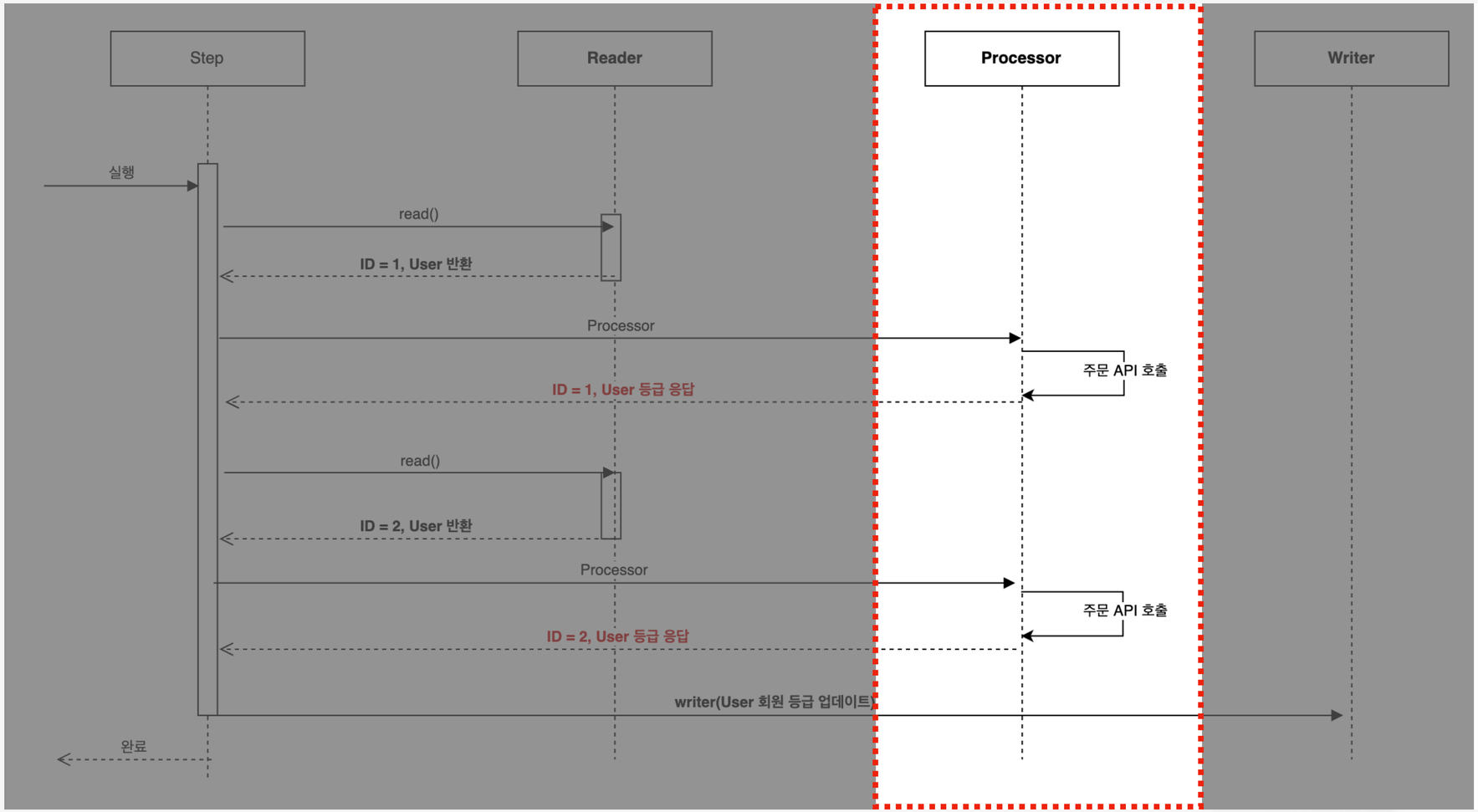
회원 등급 Batch 성능 측정

rows	기존 방식
1,000	155,880 ms
5,000	775,846 ms
10,000	1,244,827 ms
50,000	7,884,623 ms

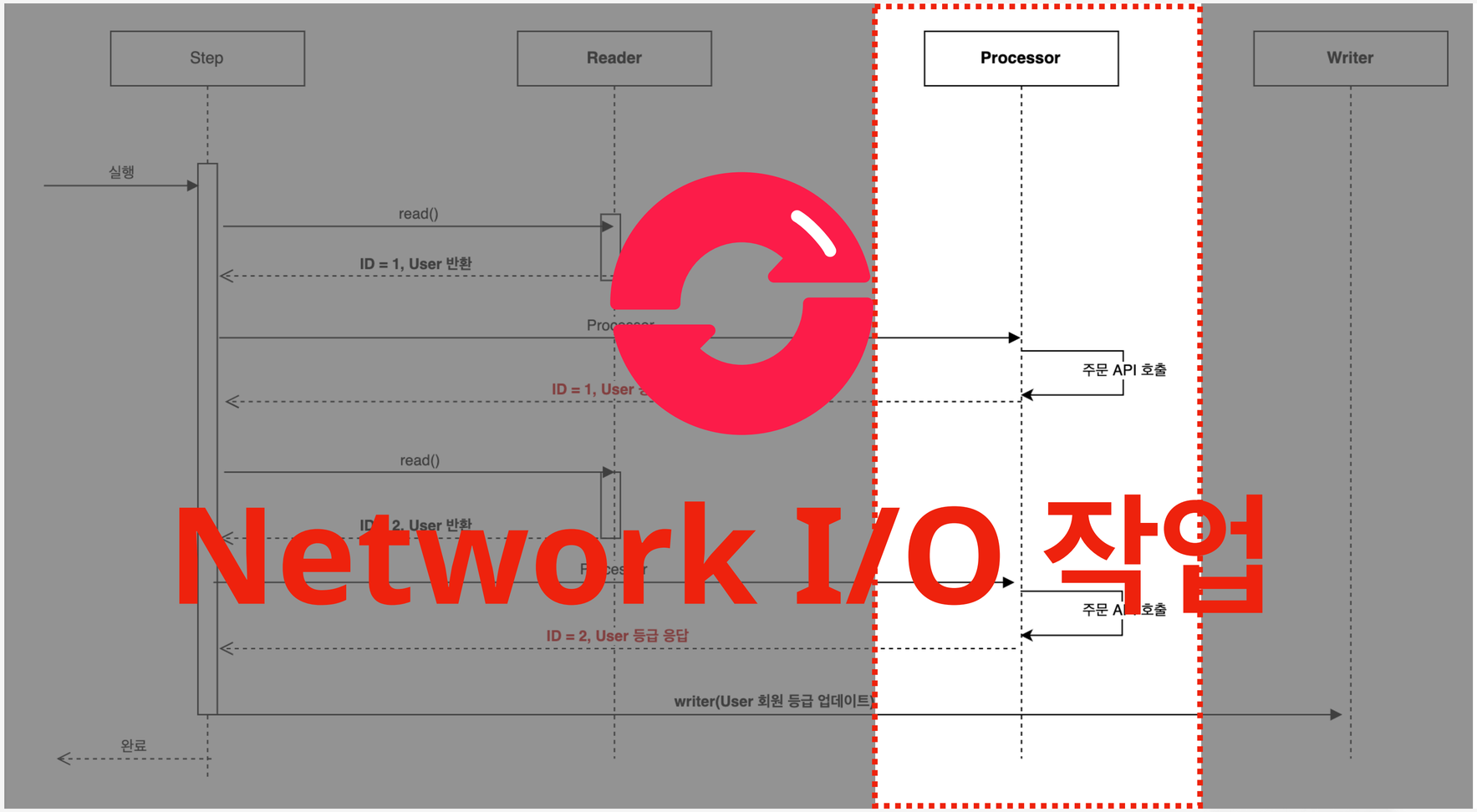
회원 등급 Batch 느린 원인



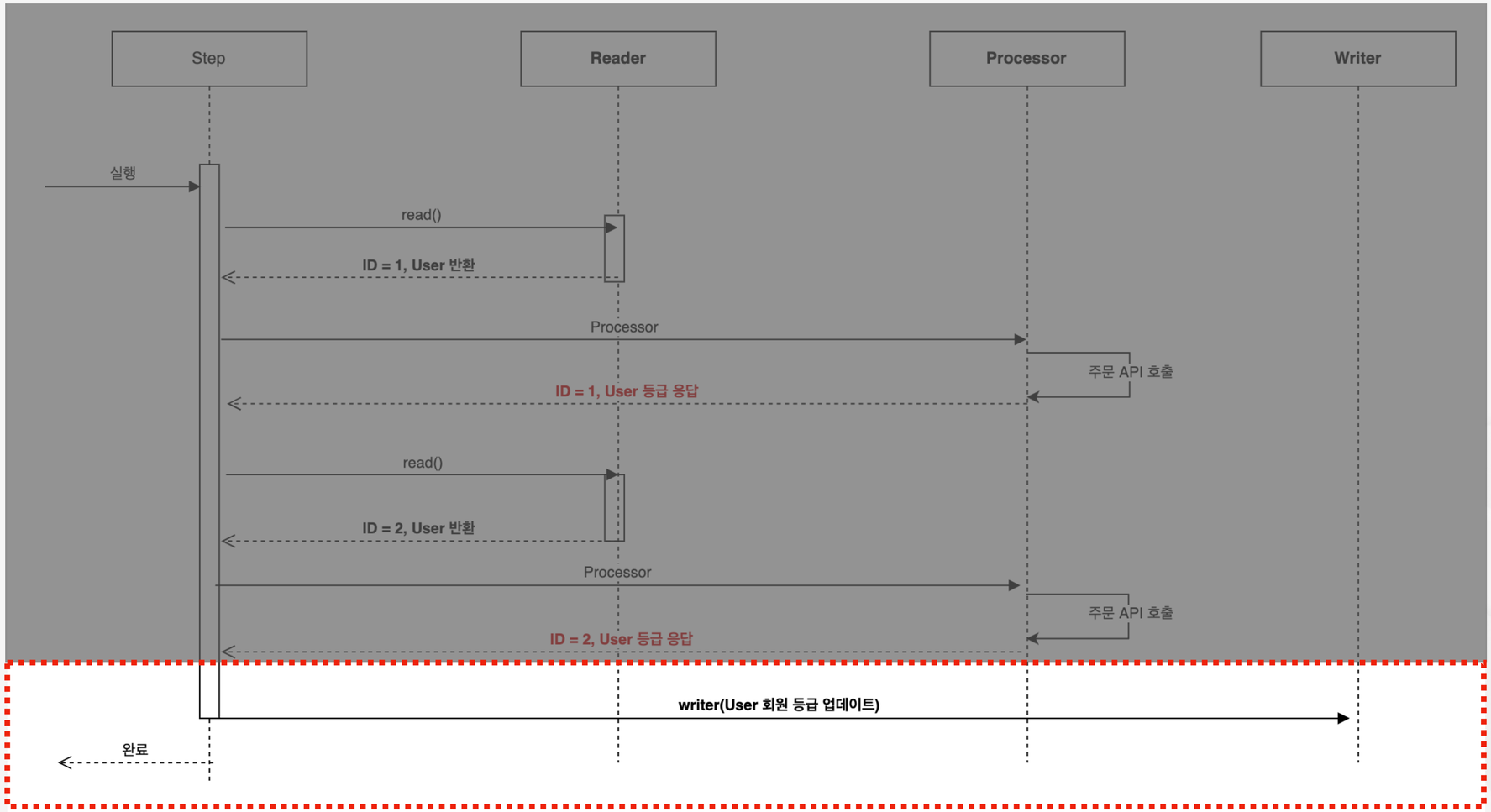
느린 원인 단일 API 호출



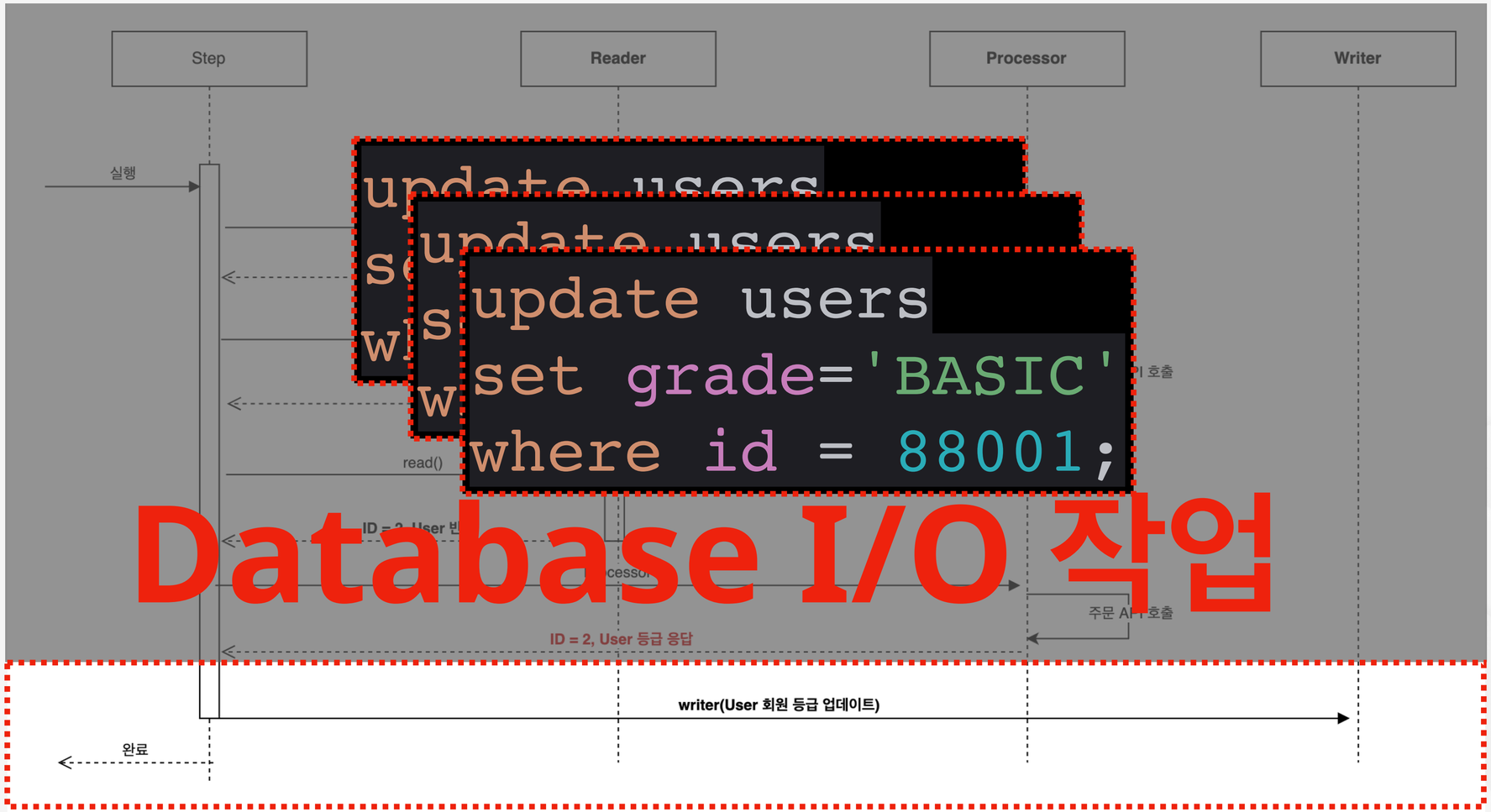
느린 원인 단일 API 호출



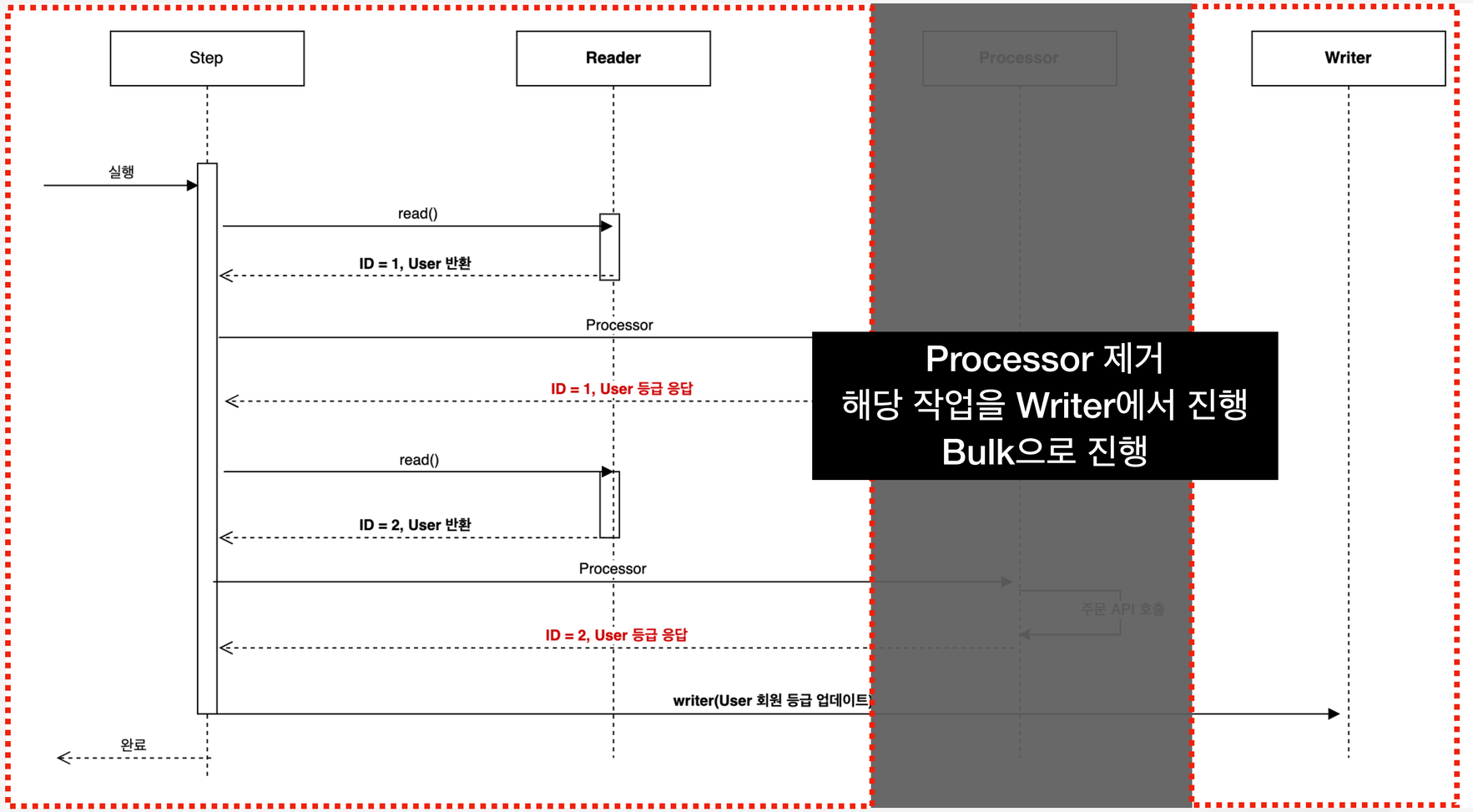
느린 원인 단건 업데이트


Step
Reader
Processor
Writer
실행
read()
ID = 1, User 반환
Processor
주문 API 호출
ID = 1, User 등급 응답
read()
ID = 2, User 반환
Processor
주문 API 호출
ID = 2, User 등급 응답
writer(User 회원 등급 업데이트)
완료

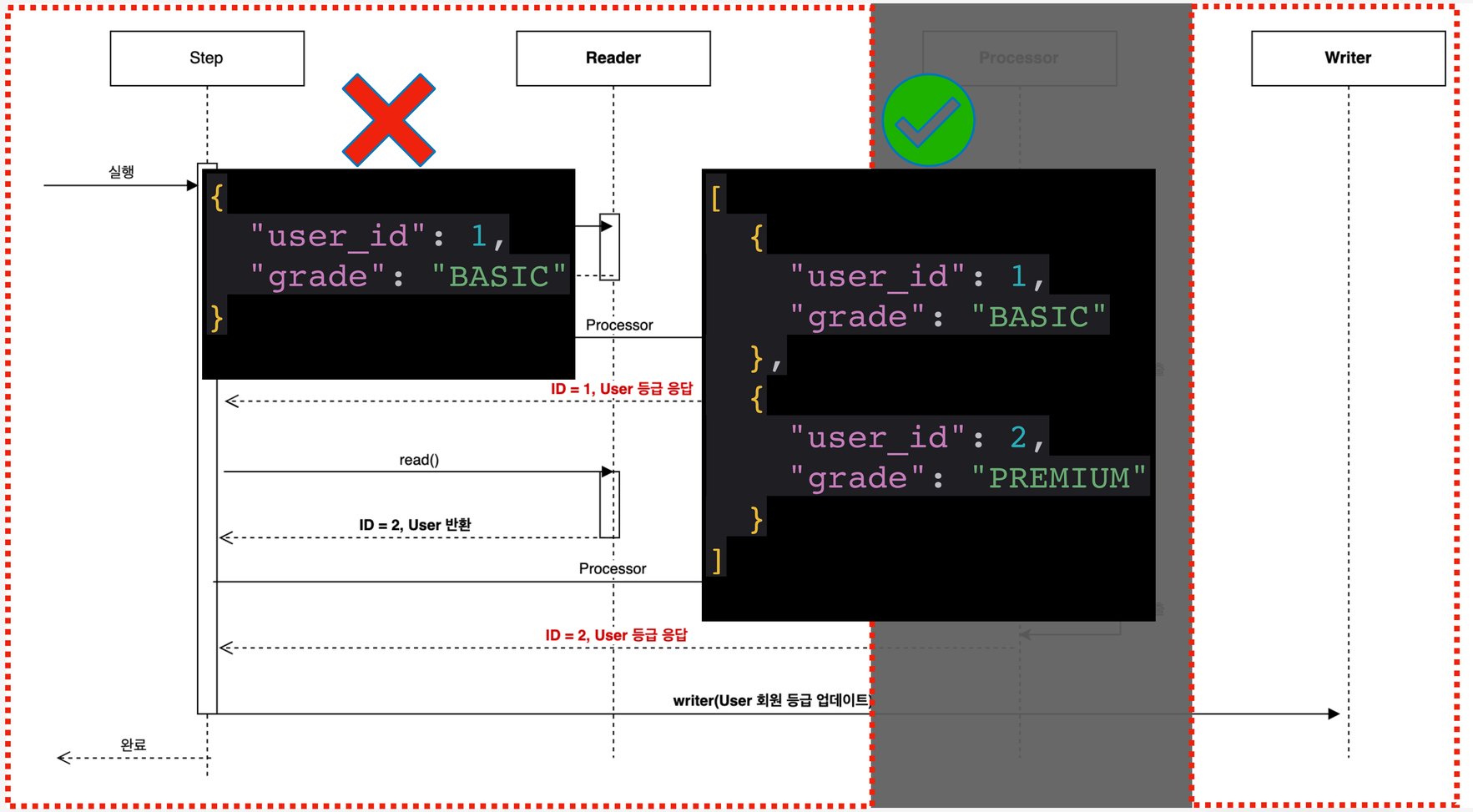
느린 원인 단건 업데이트


Step
Reader
Processor
Writer
실행
update users
set grade='BASIC'
where id = 88001;
read()
주문 API 호출
ID = 2, User 등급 응답
writer(User 회원 등급 업데이트)
완료
Database I/O 작업

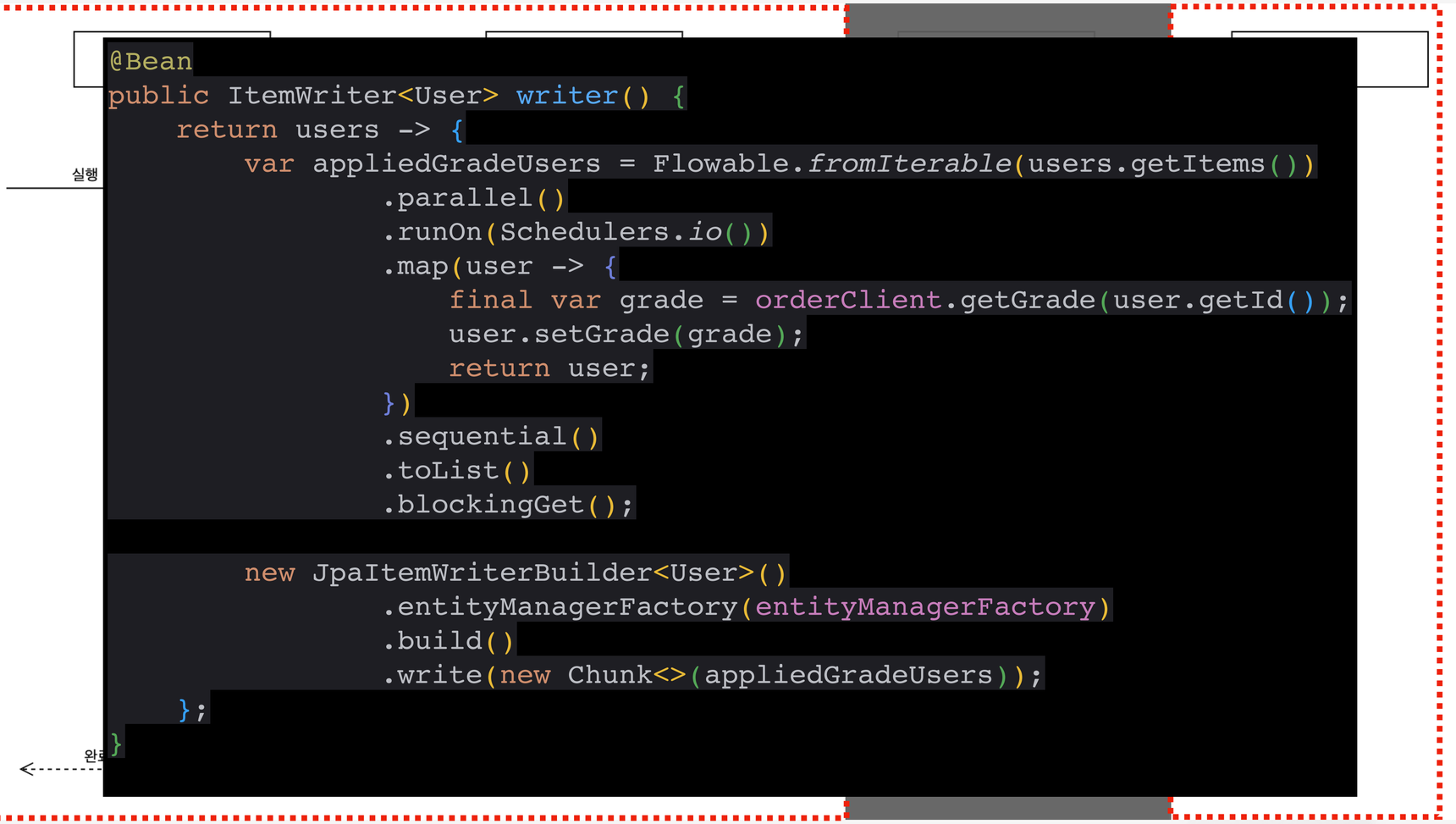
회원 등급 Batch Processor 성능 개선



회원 등급 Batch Processor 성능 개선



회원 등급 Batch Processor 성능 개선




```java
@Bean
public ItemWriter<User> writer() {
    return users -> {
        var appliedGradeUsers = Flowable.fromIterable(users.getItems())
                .parallel()
                .runOn(Schedulers.io())
                .map(user -> {
                    final var grade = orderClient.getGrade(user.getId());
                    user.setGrade(grade);
                    return user;
                })
                .sequential()
                .toList()
                .blockingGet();

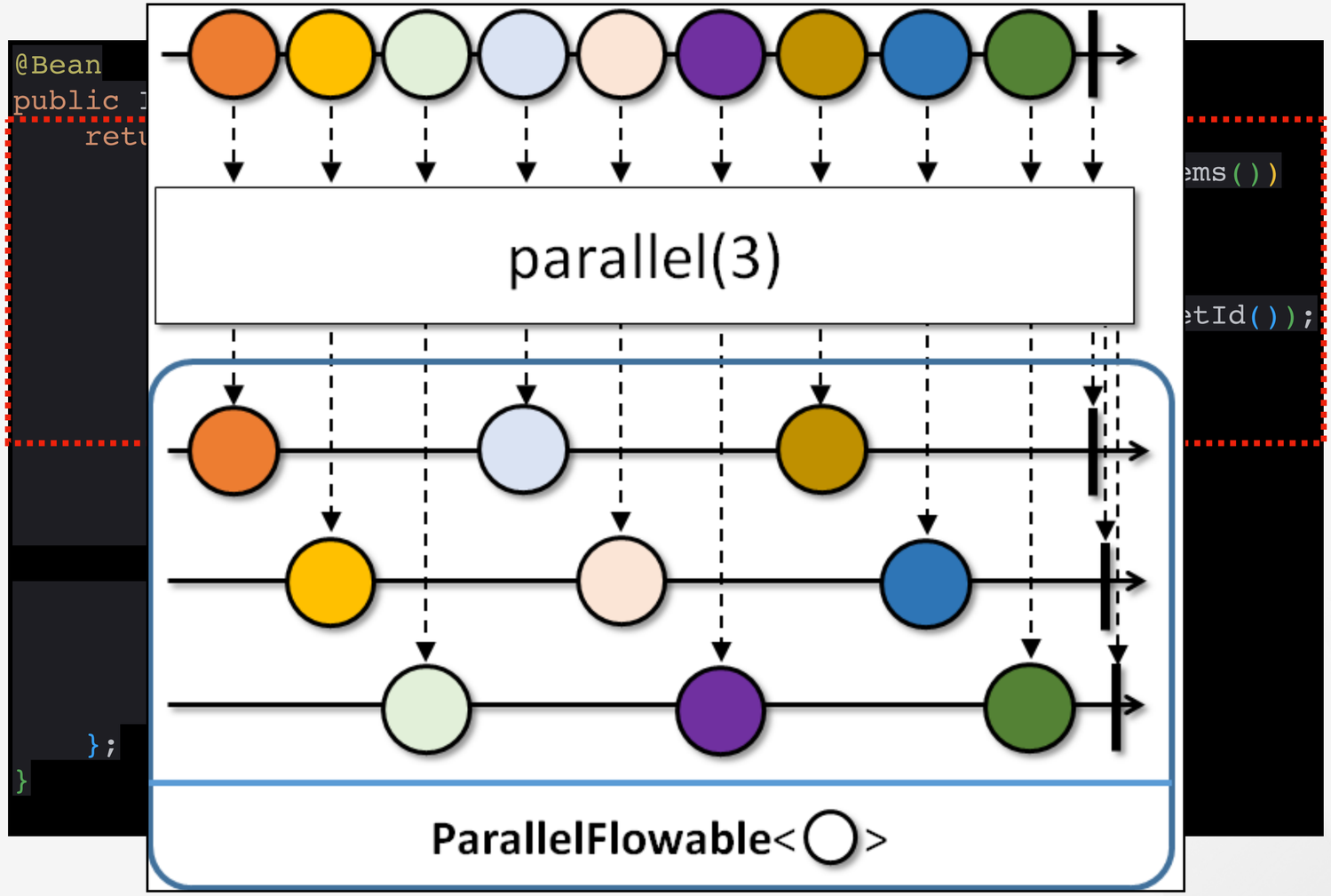
        new JpaItemWriterBuilder<User>()
                .entityManagerFactory(entityManagerFactory)
                .build()
                .write(new Chunk<>(appliedGradeUsers));
    };
}
```

회원 등급 Batch Processor 성능 개선

```
@Bean
public ItemWriter<User> writer() {
    return users -> {
        var appliedGradeUsers = Flowable.fromIterable(users.getItems())
                .parallel()
                .runOn(Schedulers.io())
                .map(user -> {
                    final var grade = orderClient.getGrade(user.getId());
                    user.setGrade(grade);
                    return user;
                })
                .sequential()
                .toList()
                .blockingGet();

        new JpaItemWriterBuilder<User>()
                .entityManagerFactory(entityManagerFactory)
                .build()
                .write(new Chunk<>(appliedGradeUsers));
    };
}
```

회원 등급 Batch Processor 성능 개선

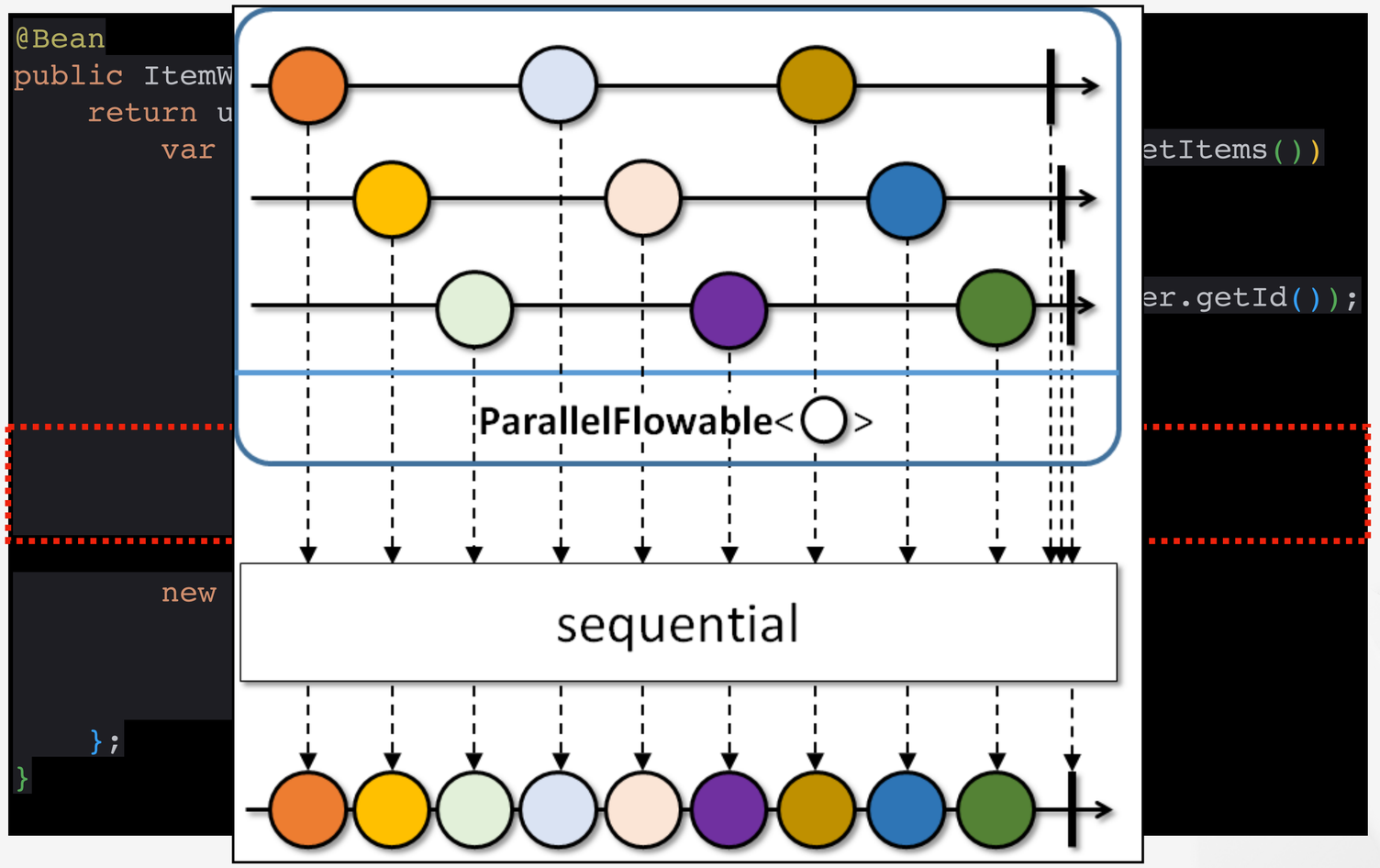


회원 등급 Batch Processor 성능 개선

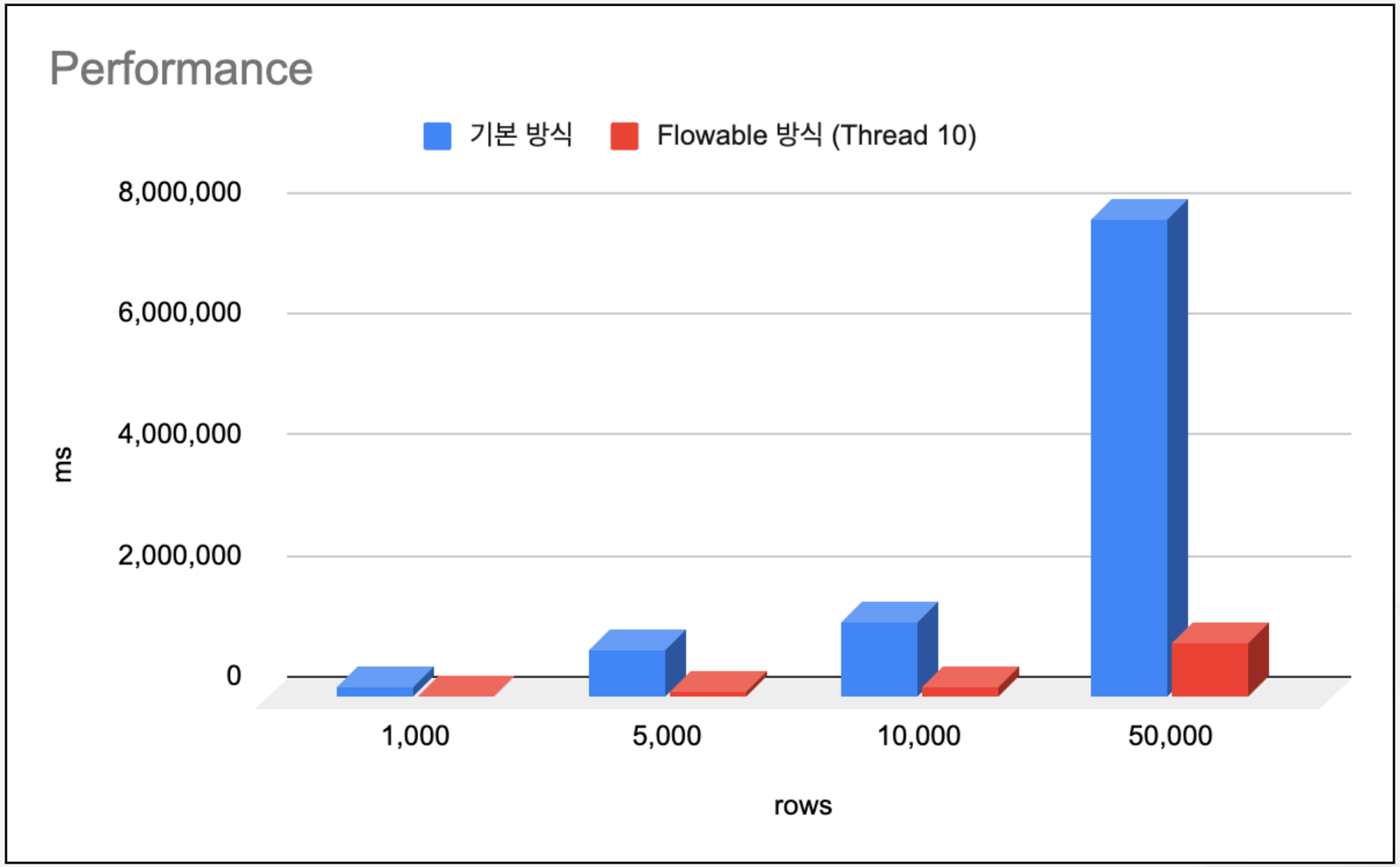
```
@Bean
public ItemWriter<User> writer() {
    return users -> {
        var appliedGradeUsers = Flowable.fromIterable(users.getItems())
                .parallel()
                .runOn(Schedulers.io())
                .map(user -> {
                    final var grade = orderClient.getGrade(user.getId());
                    user.setGrade(grade);
                    return user;
                })
                .sequential()
                .toList()
                .blockingGet();

        new JpaItemWriterBuilder<User>()
                .entityManagerFactory(entityManagerFactory)
                .build()
                .write(new Chunk<>(appliedGradeUsers));
    };
}
```

회원 등급 Batch Processor 성능 개선



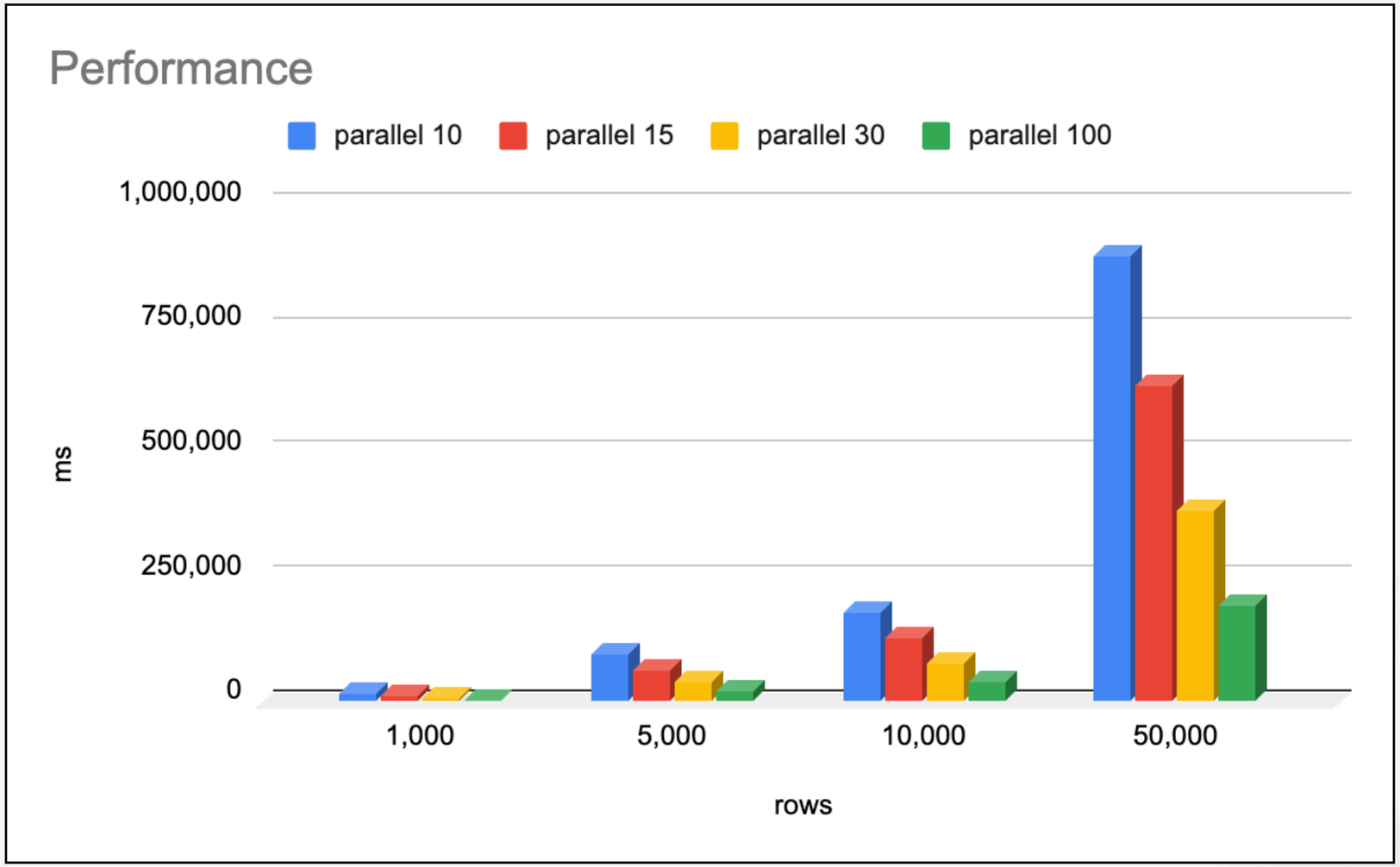
회원 등급 Batch Processor 성능 개선 결과



회원 등급 Batch 성능 측정

rows	기존 방식	Flowable 방식 (Thread 10)
1,000	155,880 ms	18,062 ms
5,000	775,846 ms	93,493 ms
10,000	1,244,827 ms	179,125 ms
50,000	7,884,623 ms	896,208 ms

병렬 처리 수준에 따른 성능 비교



병렬 처리 수준에 따른 성능 비교

rows	parallel 10	parallel 15	parallel 30	parallel 100
1,000	18,062	12,957	8,215	4,132
5,000	93,493	63,208	40,457	19,389
10,000	179,125	126,409	76,542	38,632
50,000	896,208	634,091	381,589	193,680

회원 등급 Batch Writer 성능 개선

```
@Bean
public ItemWriter<User> writer() {
    return users -> {
        var appliedGradeUsers = Flowable.fromIterable(users.getItems())
                .parallel()
                .runOn(Schedulers.io())
                .map(user -> {
                    final var grade = orderClient.getGrade(user.getId());
                    user.setGrade(grade);
                    return user;
                })
                .sequential()
                .toList()
                .blockingGet();

        new JpaItemWriterBuilder<User>()
                .entityManagerFactory(entityManagerFactory)
                .build()
                .write(new Chunk<>(appliedGradeUsers));
    };
}
```

회원 등급 Batch Writer 성능 개선

```
@Bean
public ItemWriter<User> writer() {
    return users -> {
        var appliedGradeUsers = Flowable.fromIterable(users.getItems())
Query select u1_0.id,u1_0.grade from users u1_0 where u1_0.id=88998
Query select u1_0.id,u1_0.grade from users u1_0 where u1_0.id=88999
Query select u1_0.id,u1_0.grade from users u1_0 where u1_0.id=89000
Query update users set grade='BASIC' where id=88001
Query update users set grade='VIP' where id=88002
Query update users set grade='BASIC' where id=88003
Query update users set grade='PREMIUM' where id=88004
                .entityManagerFactory(entityManagerFactory)
                .build()
    };
}
```

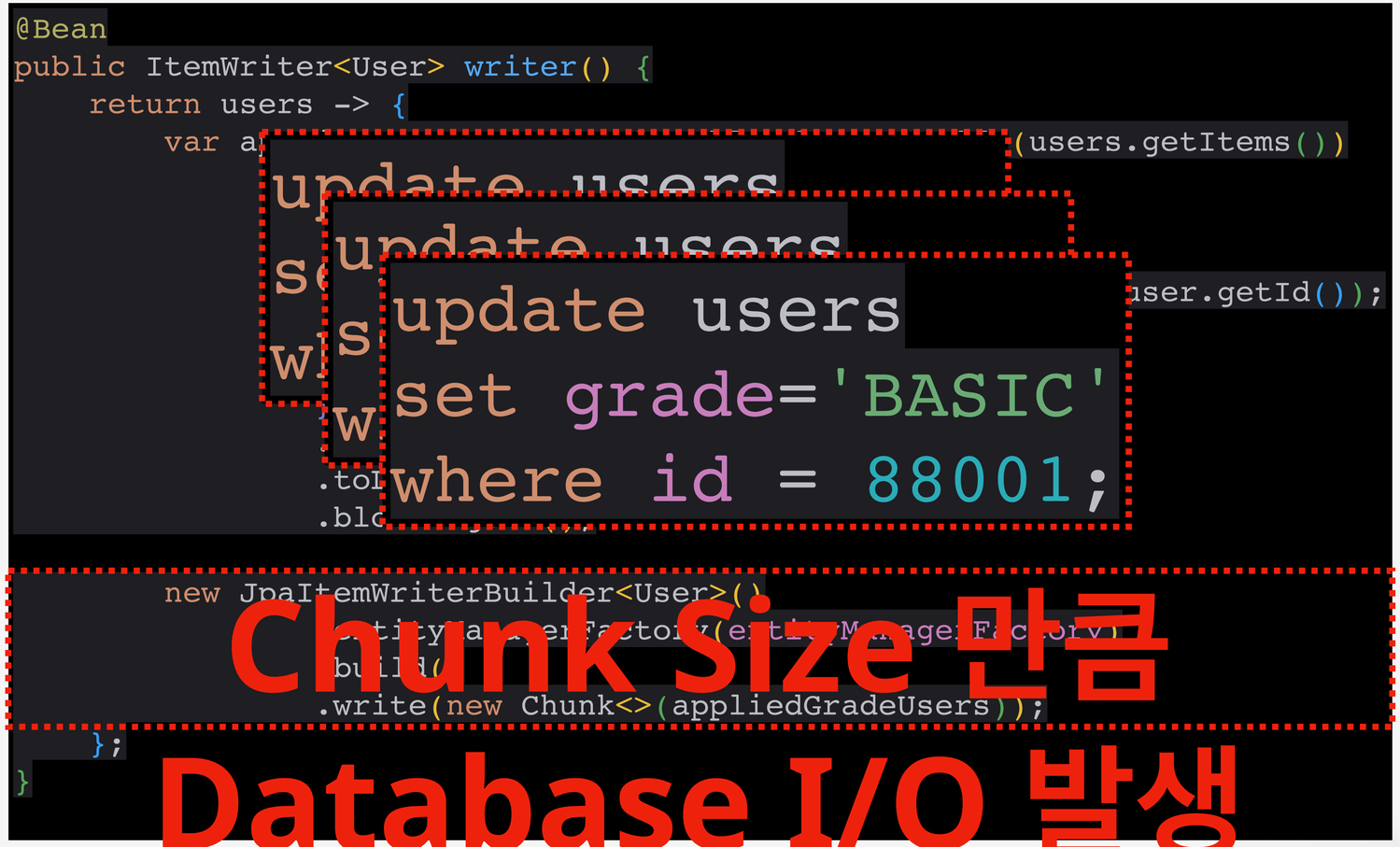
불필요한 select 쿼리 발생

회원 등급 Batch Writer 성능 개선

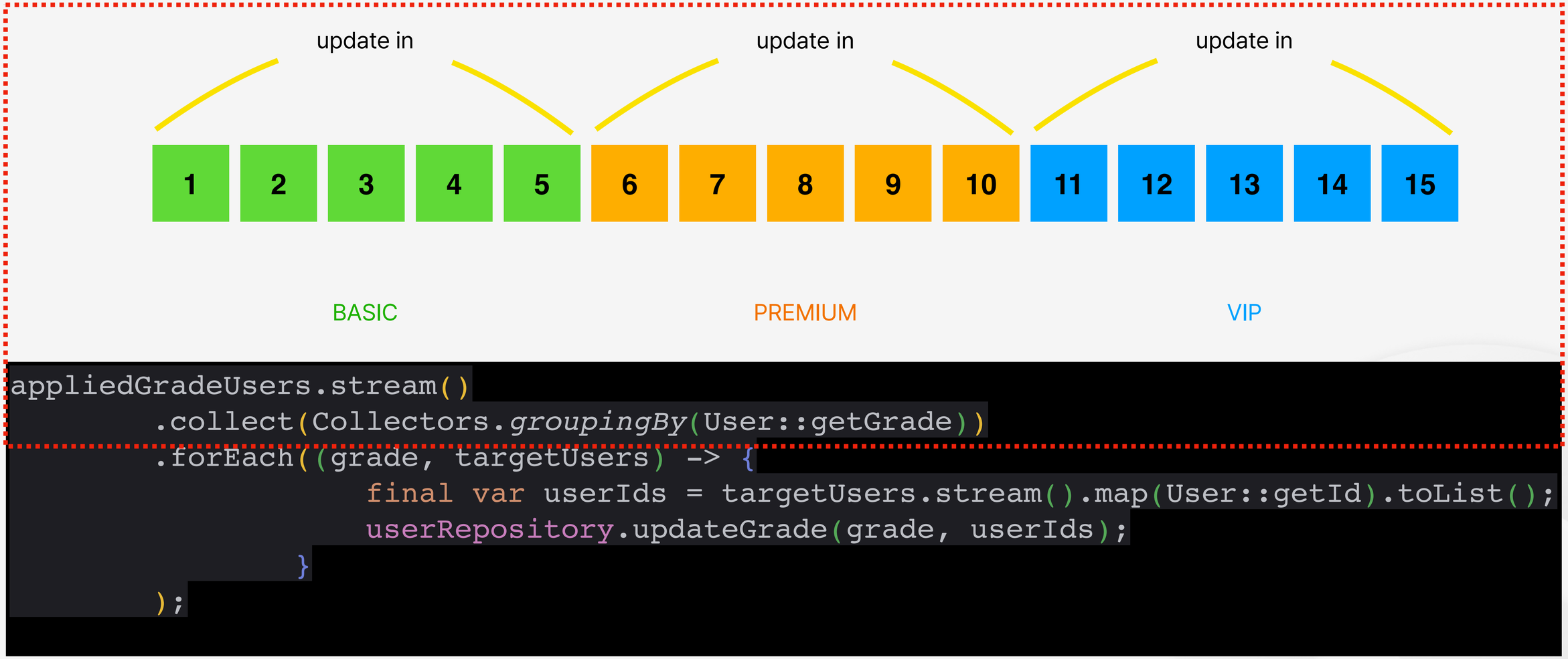
```
show variables like 'general_log%'; # general_log 획인
set global general_log = 'ON'; # `OFF` 경우 `ON` 으로 변경
```

	Variable_name	Value
1	general_log	OFF
2	general_log_file	/var/lib/mysql/24b6ff2d48b4.log

회원 등급 Batch Writer 성능 개선

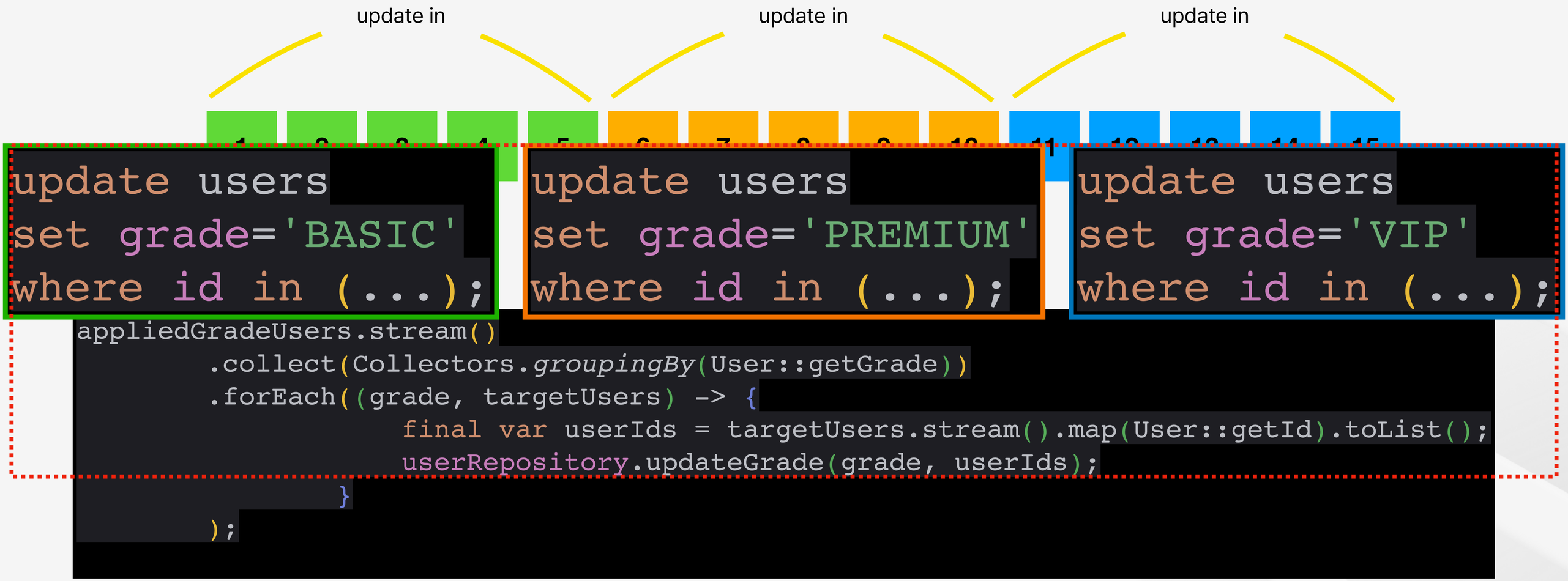


회원 등급 Batch Writer 성능 개선



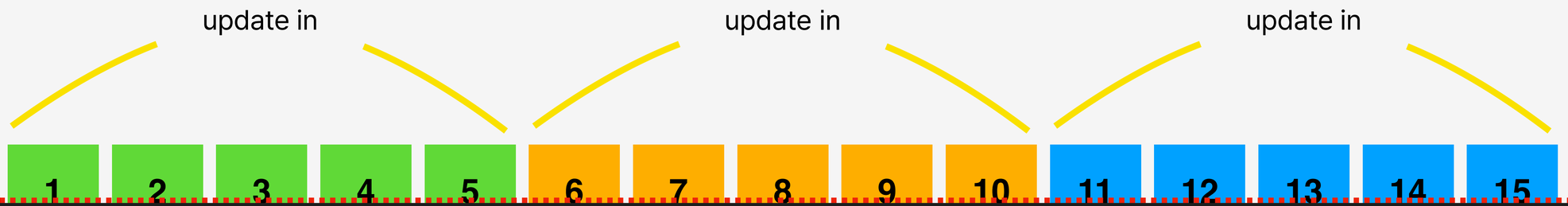

```
appliedGradeUsers.stream()
        .collect(Collectors.groupingBy(User::getGrade))
        .forEach((grade, targetUsers) -> {
                final var userIds = targetUsers.stream().map(User::getId).toList();
                userRepository.updateGrade(grade, userIds);
            }
        );
```

회원 등급 Batch Writer 성능 개선




```
appliedGradeUsers.stream()
        .collect(Collectors.groupingBy(User::getGrade))
        .forEach((grade, targetUsers) -> {
                final var userIds = targetUsers.stream().map(User::getId).toList();
                userRepository.updateGrade(grade, userIds);
        });
```

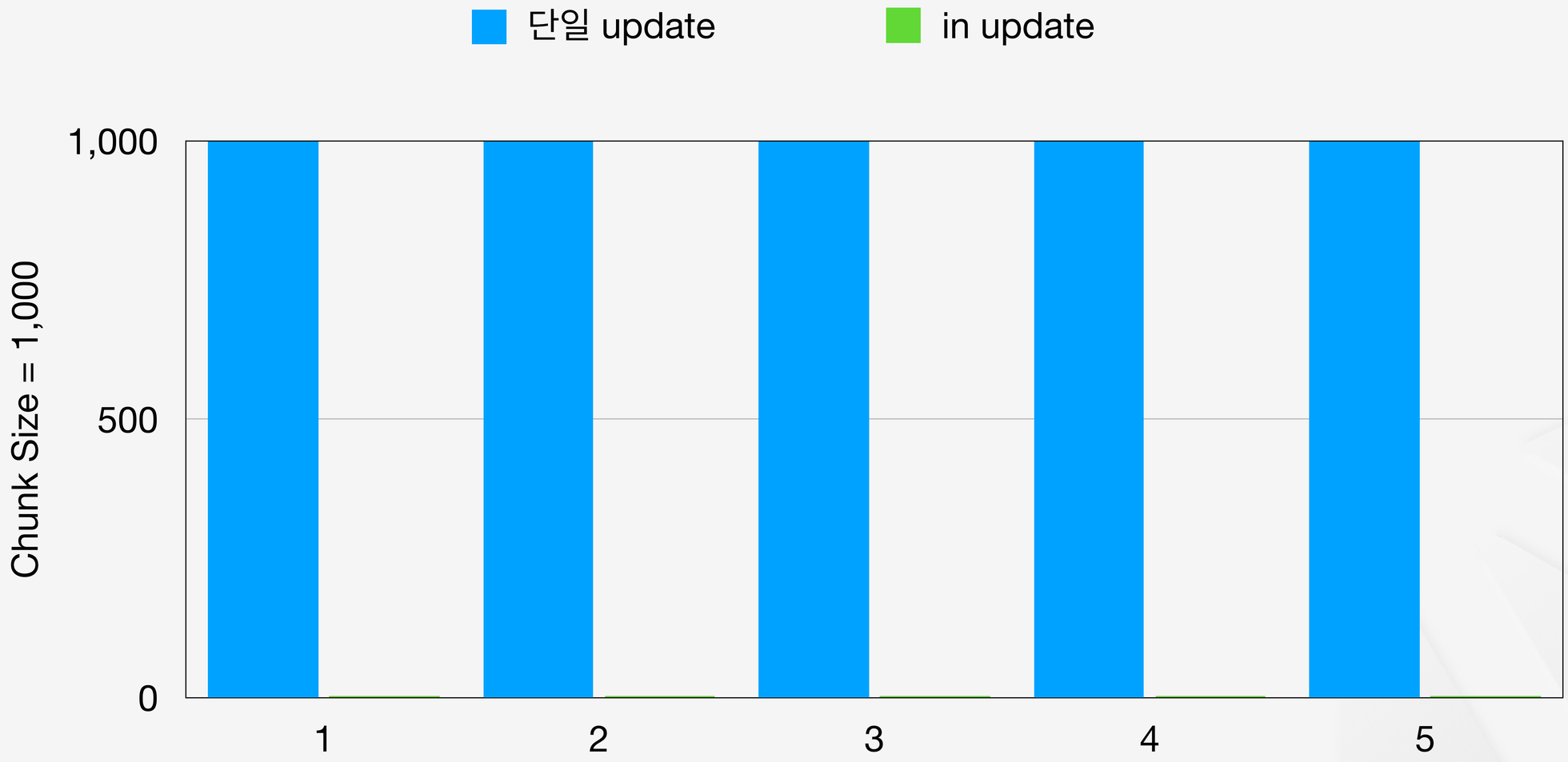
회원 등급 Batch Writer 성능 개선




```
@Transactional
@Modifying
@Query("update User u set u.grade = :grade where u.id in :ids")
```

```
        .collect(Collectors.groupingBy(User::getGrade))
        .forEach((grade, targetUsers) -> {
                final var userIds = targetUsers.stream().map(User::getId).toList();
                userRepository.updateGrade(grade, userIds);
            }
        );
```

회원 등급 Batch Writer 성능 개선



회원 등급 Batch Writer 성능 테스트 방식

```
@Test
@DisplayName("단일 방식")
void test_1() {
    // 테스트를 위해 모든 사용자를 조회합니다.
    List<User> users = userRepository.findAll();
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    for (User user : users) {
        long userId = user.getId();
        if (userId % 3 == 0) {
            user.setGrade(Grade.VIP);
        } else if (userId % 2 == 0) {
            user.setGrade(Grade.PREMIUM);
        } else {
            user.setGrade(Grade.BASIC);
        }
    }
    userRepository.saveAll(users);
    stopWatch.stop();
    System.out.println("=========================================");
    System.out.println("업데이트 소요 시간: " + stopWatch.getTotalTimeMillis() + "ms");
    System.out.println("=========================================");
}
```

회원 등급 Batch Writer 성능 테스트 방식

```
@Test
@DisplayName("update in 방식")
void test_2() {
    // 테스트를 위해 모든 사용자를 조회합니다.
    List<User> users = userRepository.findAll();
    List<Long> ids = users.stream().map(User::getId).collect(Collectors.toList());
    // 사용자 ID를 조건에 따라 그룹화하여 등급을 업데이트합니다.
    List<Long> vipIds = ids.stream().filter(id -> id % 3 == 0).collect(Collectors.toList());
    List<Long> premiumIds = ids.stream().filter(id -> id % 3 != 0 && id % 2 == 0).collect(Collectors.toList());
    List<Long> basicIds = ids.stream().filter(id -> id % 3 != 0 && id % 2 != 0).collect(Collectors.toList());
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    if (!vipIds.isEmpty()) {
        userRepository.updateGrade(Grade.VIP, vipIds);
    }
    if (!premiumIds.isEmpty()) {
        userRepository.updateGrade(Grade.PREMIUM, premiumIds);
    }
    if (!basicIds.isEmpty()) {
        userRepository.updateGrade(Grade.BASIC, basicIds);
    }
    stopWatch.stop();
    System.out.println("========================================");
    System.out.println("업데이트 소요 시간: " + stopWatch.getTotalTimeMillis() + "ms");
    System.out.println("========================================");
}
```

회원 등급 Batch Writer 성능 테스트 방식

```
-- 1,000건 데이터 삽입
CALL sp_insert_user_batch(1000);

-- 5,000건 데이터 삽입
CALL sp_insert_user_batch(5000);

-- 10,000건 데이터 삽입
CALL sp_insert_user_batch(10000);

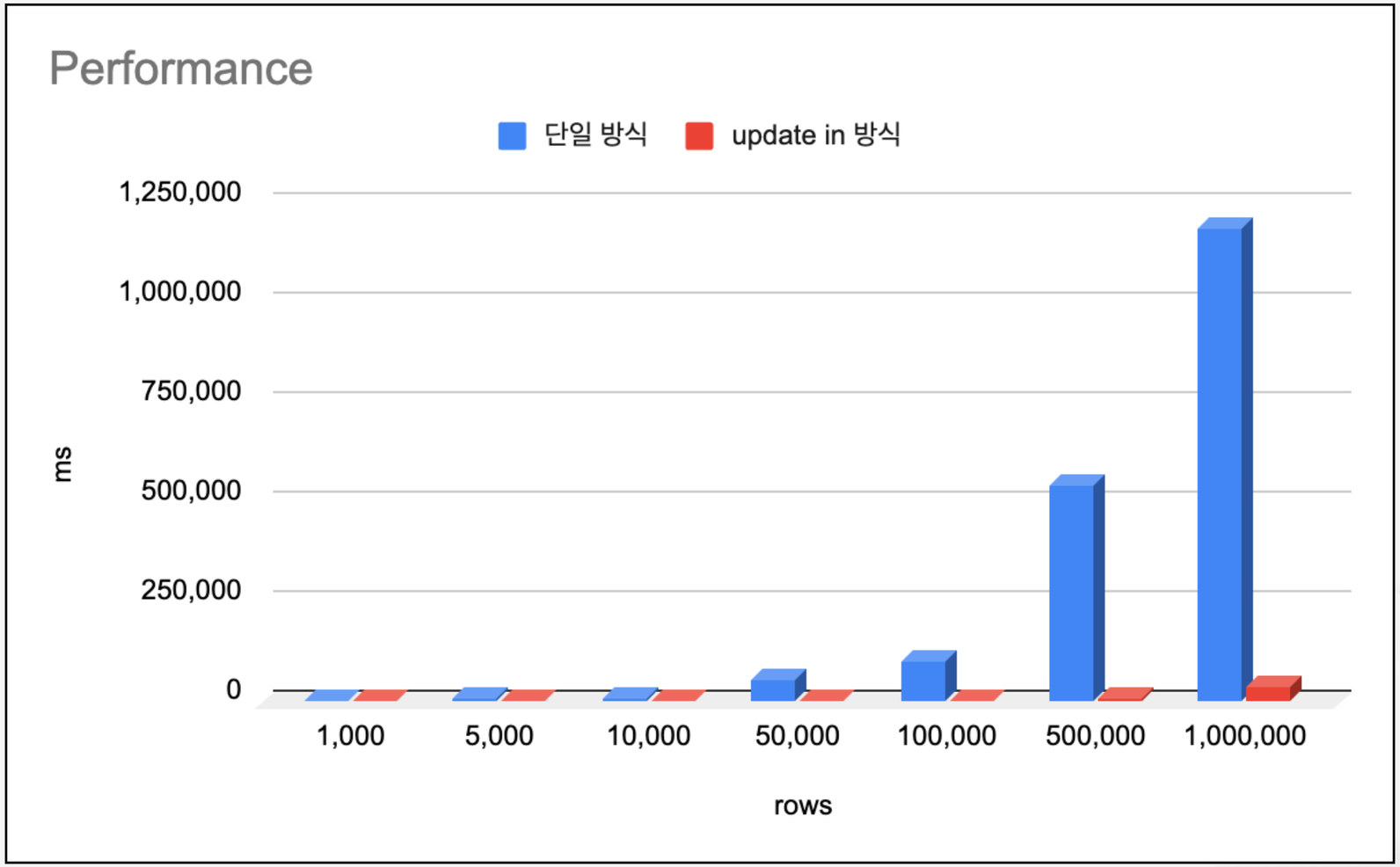
-- 50,000건 데이터 삽입
CALL sp_insert_user_batch(50000);

-- 100,000건 데이터 삽입
CALL sp_insert_user_batch(100000);

-- 500,000건 데이터 삽입
CALL sp_insert_user_batch(500000);

-- 1,000,000건 데이터 삽입
CALL sp_insert_user_batch(1000000);
```

회원 등급 Batch Writer 성능 측정



회원 등급 Batch Writer 성능 측정

rows	단일 방식	update in 방식
1,000	1,354 ms	70 ms
5,000	6,224 ms	173 ms
10,000	11,310 ms	229 ms
50,000	55,176 ms	966 ms
100,000	100,709 ms	1,865 ms
500,000	545,301 ms	10,360 ms
1,000,000	1,188,477 ms	38,688 ms

Part 2 Spring Batch 성능 최적화 기법

01

스프링 배치 성능 개선 Reader

- Limit offset 방식
- Zero offset 방식
- Cursor 방식

02

스프링 배치 성능 개선 Processor & Writer

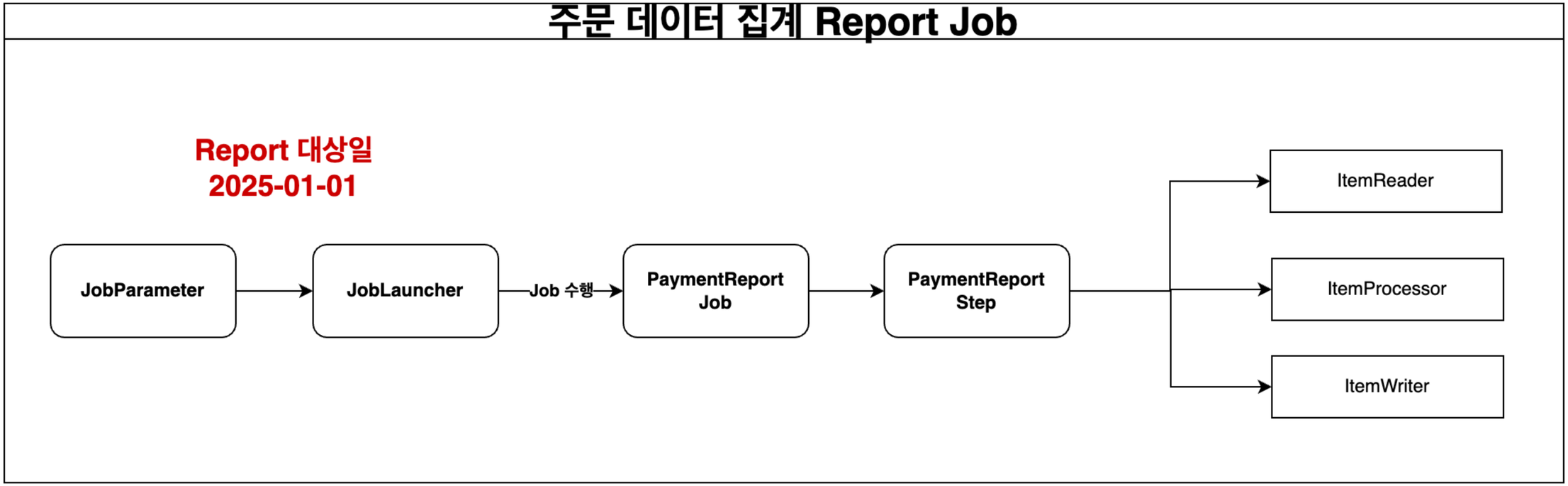
- Processor 성능 개선
- Writer 성능 개선
- 병목 원인 파악 및 개선

03

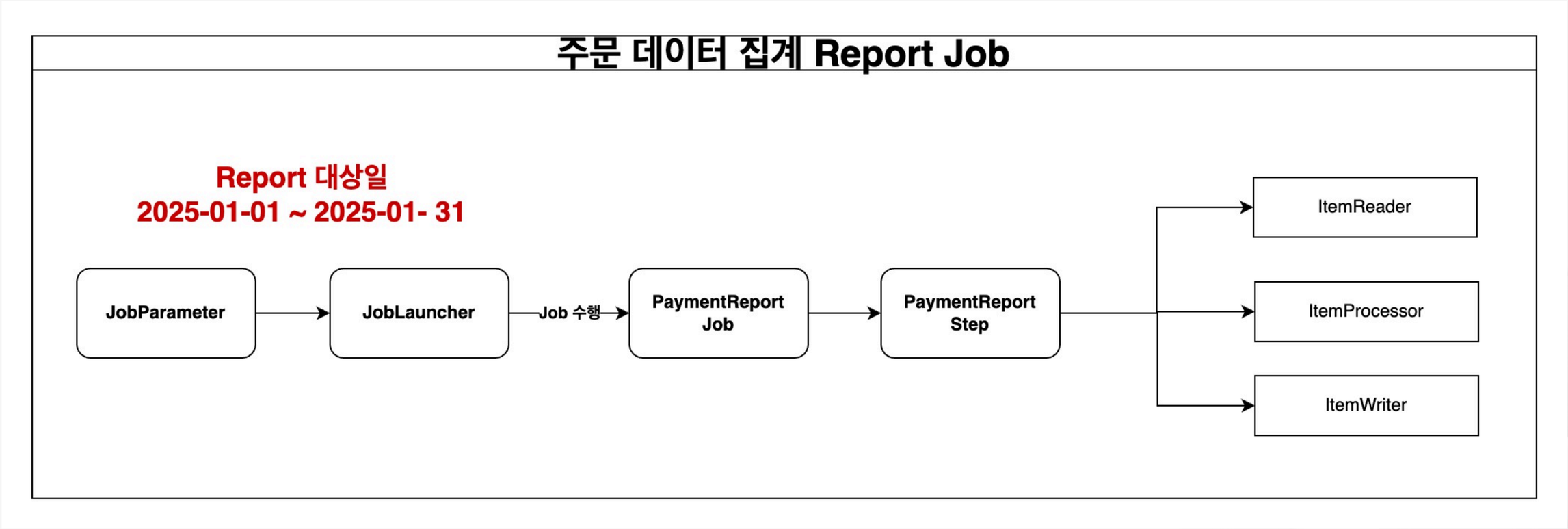
스프링 배치 성능 개선 Partition Step

- 병렬 Partition Step 처리
- MasterStep, Slave Step 수행 구조

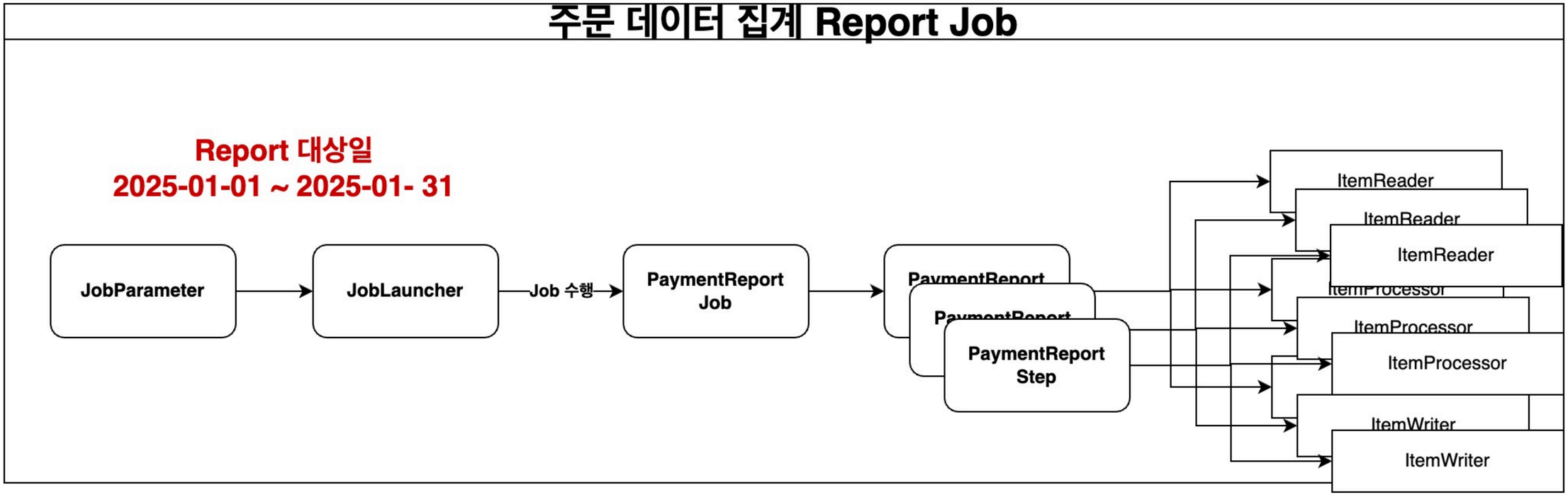
스프링 배치 성능 개선 Partitioning



스프링 배치 성능 개선 Partitioning



스프링 배치 성능 개선 Partitioning



스프링 배치 성능 개선 Partitioning

- 파티셔닝 (Partitioning) 이란?
  - 파티셔닝이란, 거대한 하나의 일감(단일 Step)을 여러 개의 작은 일감(파티션)으로 **잘게 쪼갠** 뒤, 여러 명의 작업자(스레드)에게 나누어주고 동시에 **병렬**로 처리하게 하는 스프링 배치의 가장 강력한 성능 최적화 기능입니다.
  - 결과적으로, 시스템이 가진 멀티코어 CPU 자원을 100% 활용하여 대용량 데이터 처리 시간을 획기적으로 단축시킬 수 있습니다. 수백만 건의 데이터를 처리할 때, 메모리 문제를 피하면서도 가장 빠르게 작업을 끝낼 수 있는 방법 중 하나입니다.
- 주요 사용 사례
  - **기간별 데이터 처리:** 가장 흔한 예는 기간별로 나누는 것입니다. '1년치 매출 데이터를 집계하라'는 거대한 작업 대신, "1월 데이터는 1번 작업자가, 2월 데이터는 2번. 작업자가..." 와 같이 12개의 작업으로 나누어 12개의 스레드가 동시에 처리하게 할 수 있습니다.
  - **데이터 속성별 분할:** 데이터의 특정 속성을 기준으로 나눌 수도 있습니다. 예를 들어, '모든 사용자의 등급을 갱신하라'는 작업을 "VIP 등급은 1번 작업자, PREMIUM 등급은 2번 작업자..." 와 같이 사용자 등급별로 작업을 분할하여 병렬 처리할 수 있습니다.
  - **대용량 파일 처리**: 수십 기가바이트(GB)에 달하는 거대한 텍스트 파일이 있다면, 이 파일을 행(line) 번호를 기준으로 여러 조각으로 나눌 수 있습니다. "1번 작업자는 1~100만 번째 줄, 2번 작업자는 100만 1~200만 번째 줄..." 과 같이 나누어 동시에 파일을 읽고 처리하는 것입니다.

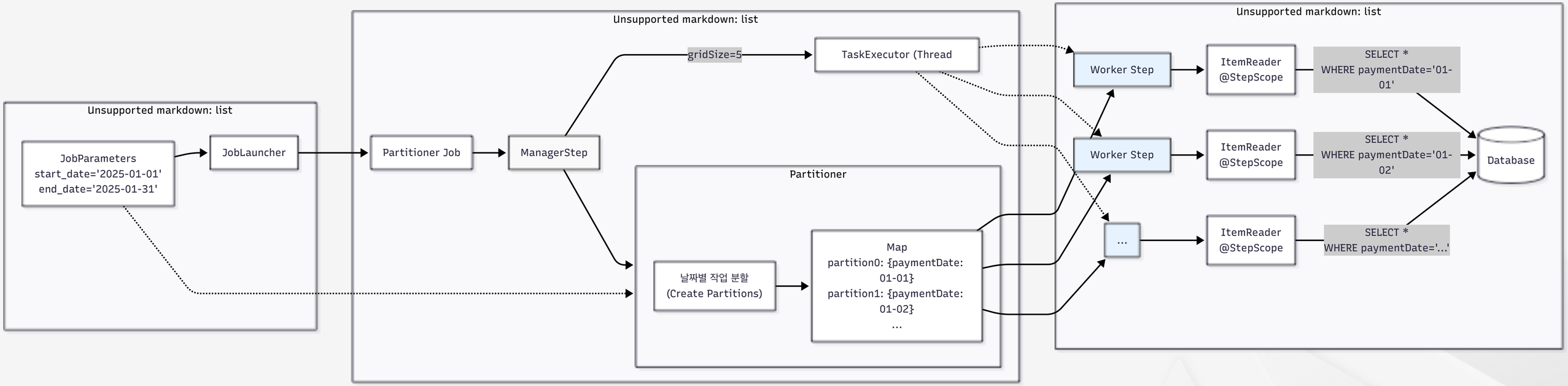
스프링 배치 성능 개선 Partitioning

- 파티셔닝 아키텍처
  - 파티셔닝 아키텍처는 작업을 **분할하고 지시하는 '매니저 스텝'과**, 지시를 받아 **실제 데이터를 처리하는 '워커 스텝'**이라는 두 가지 명확한 역할로 구성됩니다.
- 매니저 스텝 (Manager Step)
  - 매니저 스텝은 파티셔닝의 컨트롤 타워입니다. 직접 데이터를 처리하지 않고, 전체 작업을 분할하고 워커 스텝들을 관리 및 실행하는 책임을 가집니다.
- 주요 책임
  - 파티션 생성 (Partitioner): **Partitioner**는 전체 데이터 범위를 어떻게 나눌지 결정하는 로직을 담은 컴포넌트입니다. JobParameters를 참고하여, 각 워커가 처리할 데이터의 범위(예: 날짜, ID 범위)를 **ExecutionContext**에 담아 여러 개의 파티션을 생성합니다.
  - 워커 스텝 실행: 생성된 파티션의 수만큼 워커 스텝을 실행하도록 스케줄링합니다.
  - 병렬 처리 (TaskExecutor): TaskExecutor (스레드 풀)를 사용하여 워커 스텝들을 병렬로 동시에 실행시킵니다. gridSize 옵션으로 동시에 실행할 워커의 최대 개수를 제어할 수 있습니다.

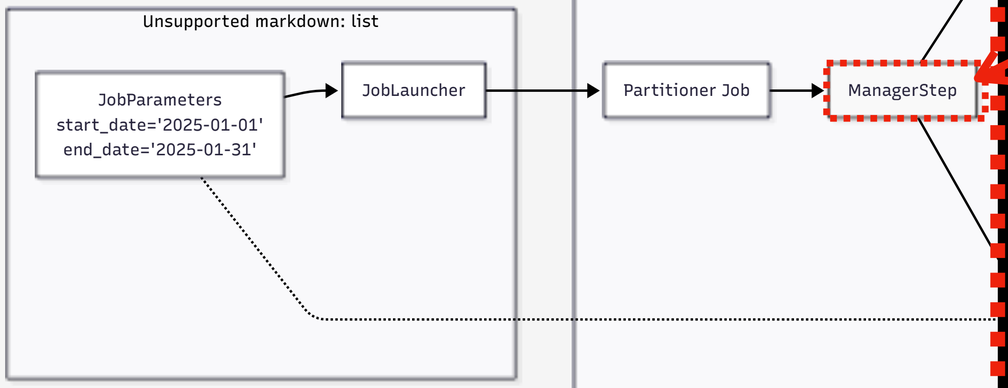
스프링 배치 성능 개선 Partitioning

- 워커 스텝 (Worker Step)
  - 워커 스텝은 매니저로부터 특정 데이터 처리 임무를 할당받은 실제 작업 단위입니다. 각 워커는 자신에게 주어진 범위의 데이터만 독립적으로 처리합니다.
- 주요 책임
  - 데이터 수신: 각 워커 스텝은 시작될 때, 매니저가 파티셔너를 통해 생성한 **StepExecutionContext**를 전달받습니다. 코드에서는 **@StepScope**와 @Value 어노테이션을 사용하여 이 컨텍스트에 담긴 파라미터(예: 처리할 날짜)를 주입받습니다.
  - 독립적 실행: 각 워커는 독립적인 Reader, Processor, Writer 인스턴스를 가집니다. 이를 통해 다른 워커의 **작업에 전혀 영향을 주지 않고 자신만의 데이터 처리**를 수행합니다.
  - 트랜잭션 관리: 각 워커 스텝은 자신만의 트랜잭션 범위 내에서 동작합니다. 따라서 특정 워커에서 오류가 발생하여 롤백되더라도, 다른 워커들의 트랜잭션에는 영향을 미치지 않아 작업의 격리성을 보장합니다.

스프링 배치 성능 개선 Partitioning

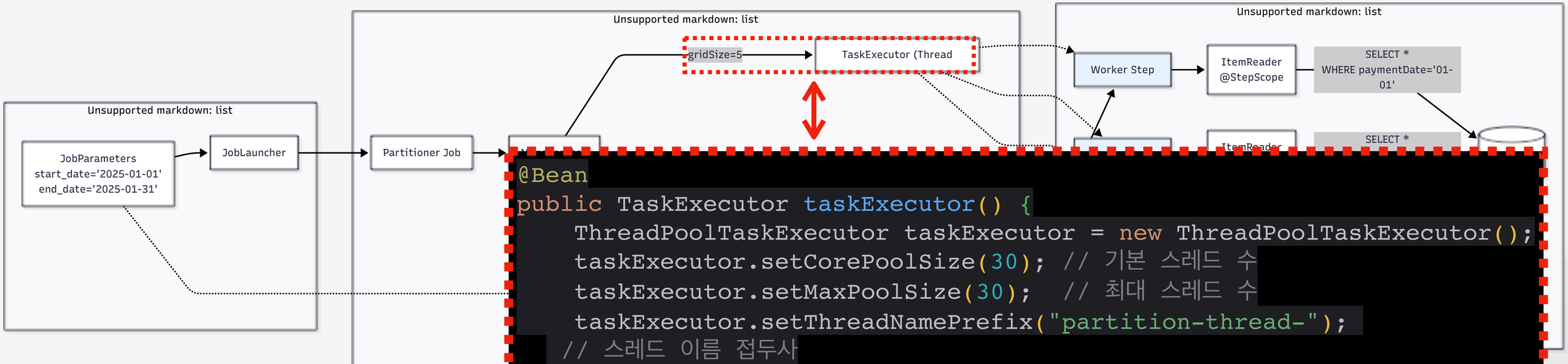


스프링 배치 성능 개선 Partitioning




```java
@Bean
public Step managerStep(Step workerStep) {
    return new StepBuilder("managerStep", jobRepository)
            // "workerStep"이라는 이름으로 파티셔너를 설정합니다.
            .partitioner("workerStep", partitioner(null, null))
            // 각 파티션에서 실행될 스텝을 지정합니다.
            .step(workerStep)
            // 동시에 실행할 파티션의 수를 설정합니다.
            .gridSize(5)
            // 파티션을 병렬로 처리하기 위한 TaskExecutor를 설정합니다.
            .taskExecutor(taskExecutor())
            .build();
}
```

스프링 배치 성능 개선 Partitioning




```java
@Bean
public TaskExecutor taskExecutor() {
    ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
    taskExecutor.setCorePoolSize(30); // 기본 스레드 수
    taskExecutor.setMaxPoolSize(30);  // 최대 스레드 수
    taskExecutor.setThreadNamePrefix("partition-thread-");
    // 스레드 이름 접두사
    taskExecutor.setWaitForTasksToCompleteOnShutdown(true);
    // 종료 시 모든 태스크가 완료될 때까지 대기
    taskExecutor.initialize();
    return taskExecutor;
}
```

스프링 배치 성능 개선 Partitioning

```java
@Override
public Map<String, ExecutionContext> partition(int
gridSize) {
    Map<String, ExecutionContext> partitions = new
HashMap<>();
    LocalDate currentDate = startDate;
    int partitionIndex = 0;

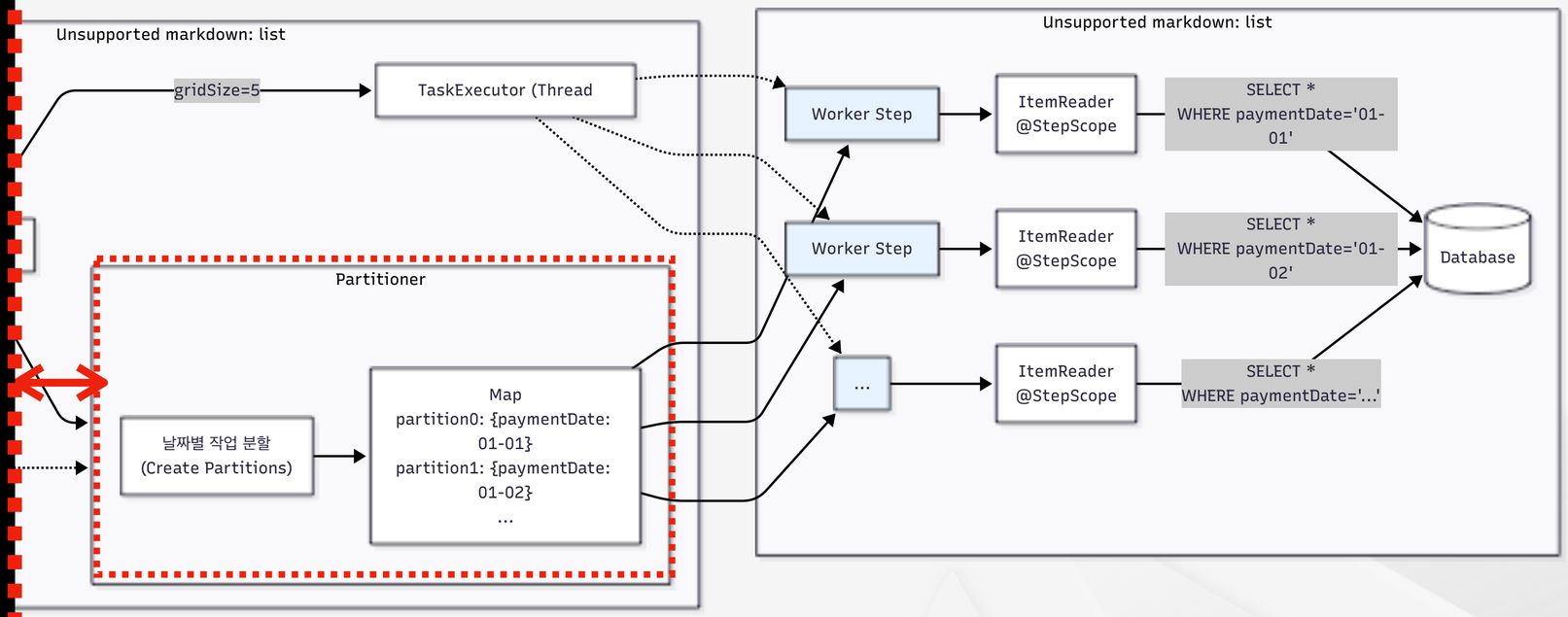
    // 시작일부터 종료일까지 반복하여 파티션 생성
    while (!currentDate.isAfter(endDate)) {
        ExecutionContext context = new
ExecutionContext();
        // "paymentDate" 키에 현재 날짜를 ISO_LOCAL_DATE 형식
의 문자열로 저장

context.putString(EXECUTION_CONTEXT_KEY_PAYMENT_DATE,

currentDate.format(DateTimeFormatter.ISO_LOCAL_DATE));
        // "partition" + 인덱스 형태의 키로 파티션 추가
        partitions.put(PARTITION_KEY + partitionIndex,
context);
        …

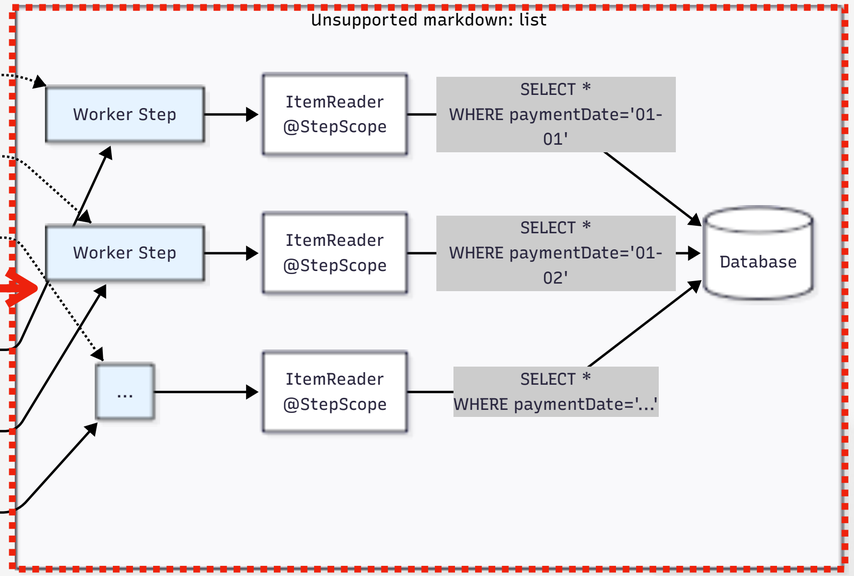
        // 다음 날짜로 이동
        currentDate = currentDate.plusDays(1);
        partitionIndex++;
    }

    return partitions;
}
```

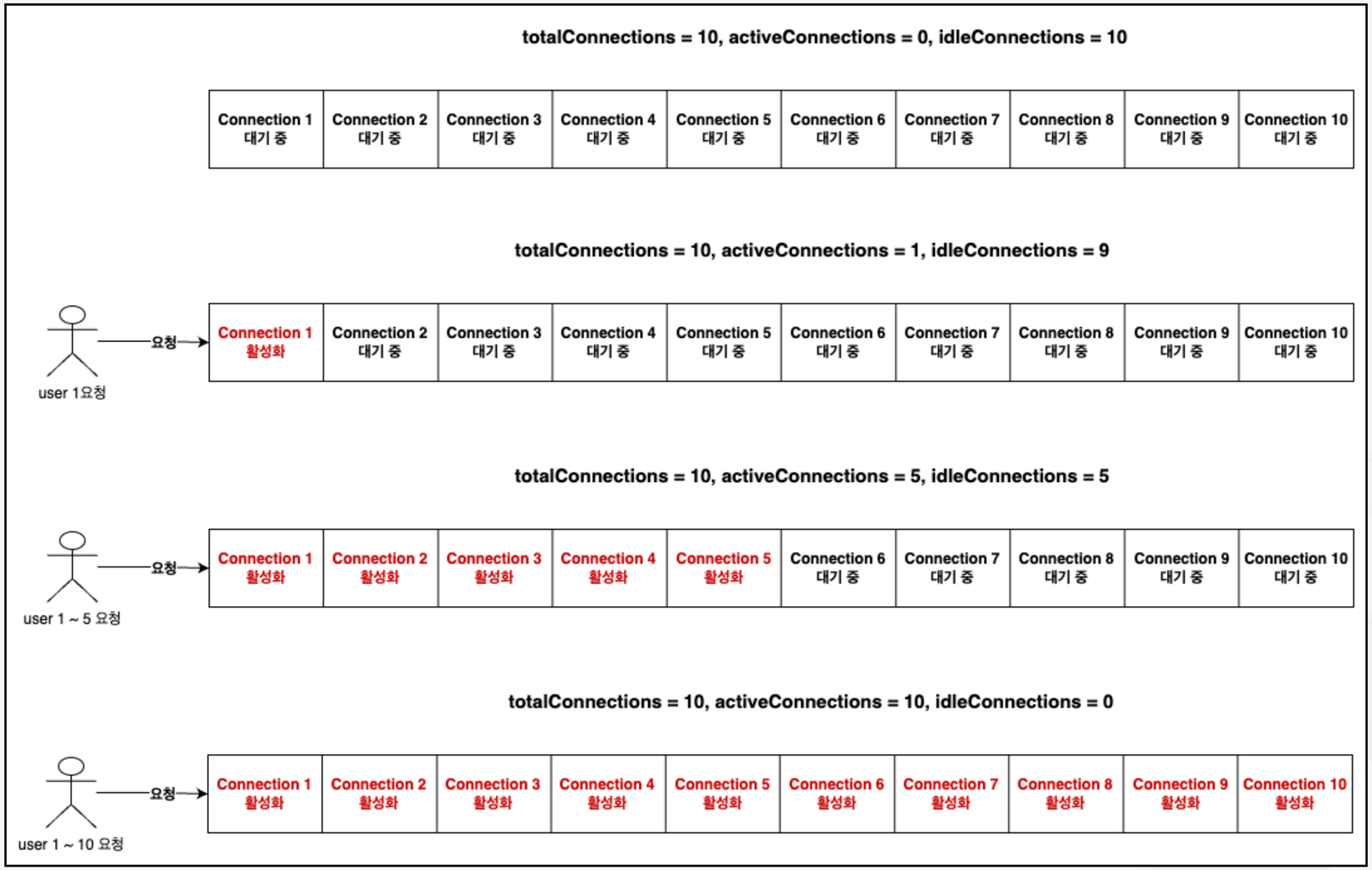


스프링 배치 성능 개선 Partitioning

```java
@Bean
public Step workerStep(
        JpaCursorItemReader<PaymentSource> cursorItemReader
) {
    return new StepBuilder("workerStep", jobRepository)
            .<PaymentSource, Payment>chunk(chunkSize, transactionManager)
            .listener(new StepDurationTrackerListener())
            .reader(cursorItemReader)
            .processor(paymentReportProcessor())
            .writer(paymentReportWriter())
            .listener(new ChunkDurationTrackerListener())
            .build();
}
```



커넥션 풀 시나리오


totalConnections = 10, activeConnections = 0, idleConnections = 10
Connection 1 대기 중
Connection 2 대기 중
Connection 3 대기 중
Connection 4 대기 중
Connection 5 대기 중
Connection 6 대기 중
Connection 7 대기 중
Connection 8 대기 중
Connection 9 대기 중
Connection 10 대기 중
totalConnections = 10, activeConnections = 1, idleConnections = 9
요청
user 1요청
Connection 1 활성화
totalConnections = 10, activeConnections = 5, idleConnections = 5
요청
user 1 ~ 5 요청
Connection 1 활성화
Connection 2 활성화
Connection 3 활성화
Connection 4 활성화
Connection 5 활성화
totalConnections = 10, activeConnections = 10, idleConnections = 0
요청
user 1 ~ 10 요청
Connection 1 활성화
Connection 2 활성화
Connection 3 활성화
Connection 4 활성화
Connection 5 활성화
Connection 6 활성화
Connection 7 활성화
Connection 8 활성화
Connection 9 활성화
Connection 10 활성화

스프링 배치 성능 개선 Partitioning

gridSize	taskExecutor maxPoolSize	dataSource maximum-pool-size	결과
30	30	30	(이상적) 30개의 워커가 30개 스레드에서 원활히 동작하며, 각자 DB 커넥션을 문제없이 사용.
30	10	30	(스레드 병목) gridSize가 30이라도 실제 스레드는 10개뿐. 10개 워커만 동시에 실행되고 나머지는 대기. 실제 동시성은 10.
30	30	10	(커넥션 병목 / 오류 발생) 30개 스레드가 동시에 실행되어 각자 DB 커넥션을 요청. 커넥션 풀에는 10개만 있으므로, 11번째 요청부터는 커넥션을 얻지 못하고 대기하다가 Connection is not available 오류 발생. (현재 겪고 계신 문제의 원인)

Part 3: Spring Batch 실무에서의 배치

Part 3 Spring Batch 실무에서의 배치

01

재처리가 가능한 Batch

- 멱등성 배치 어플리케이션 개발
- 데이터 누락이 케이스 대응

02

HTTP Page Item Reader

- 외부 데이터를 읽어 오는 방법
- 시스템의 강결합 해결

03

Batch Test 작성하는 노하우

- 다양한 테스트 케이스를 확보 하기 위한 방법
- 실무에서 의미 있는 테스트 코드를 작성하는 방법

Part 3 Spring Batch 실무에서의 배치

01

재처리가 가능한 Batch

- 멱등성 배치 어플리케이션 개발
- 데이터 누락이 케이스 대응

02

HTTP Page Item Reader

- 외부 데이터를 읽어 오는 방법
- 시스템의 강결합 해결

03

Batch Test 작성하는 노하우

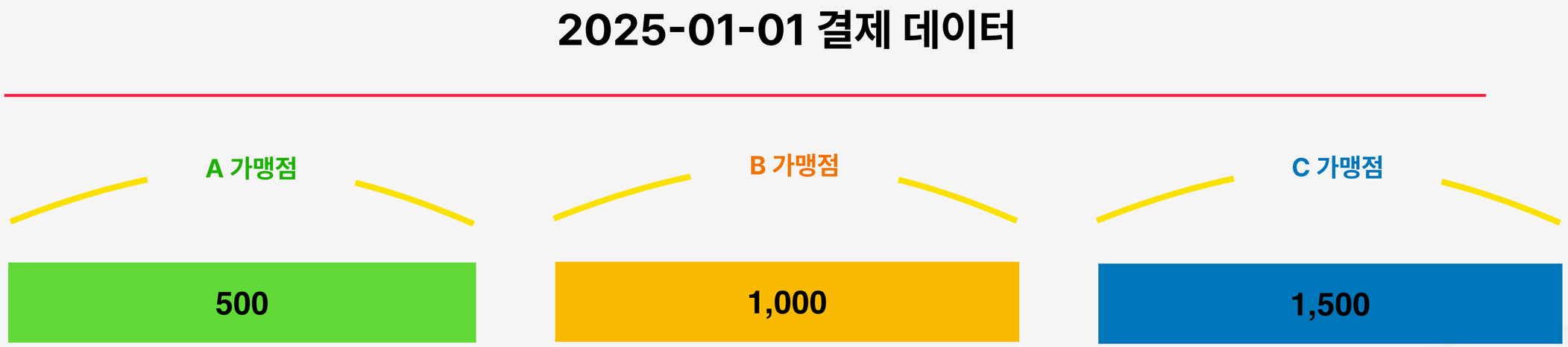
- 다양한 테스트 케이스를 확보 하기 위한 방법
- 실무에서 의미 있는 테스트 코드를 작성하는 방법

재처리가 가능한 Batch 시나리오

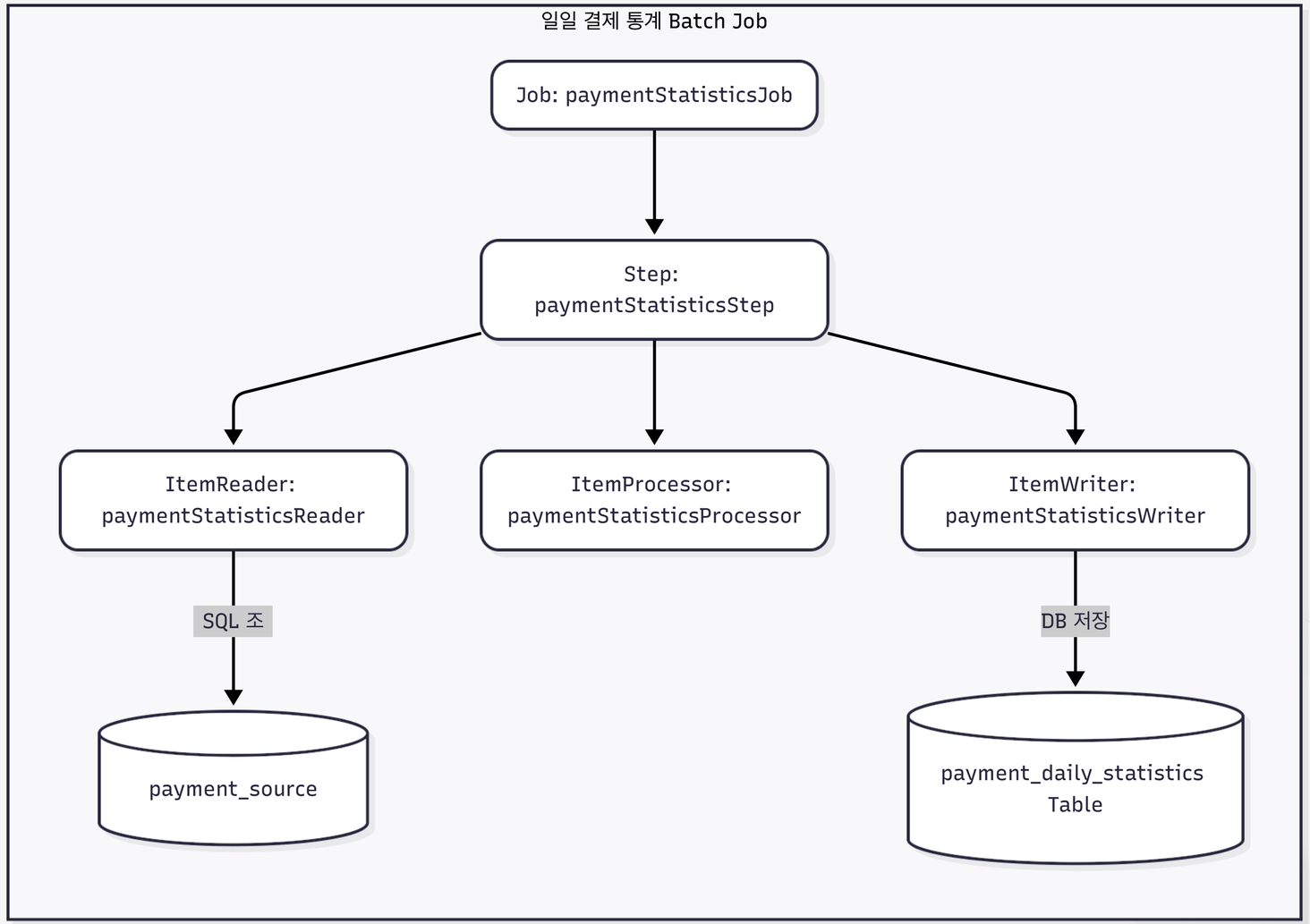
2025-01-01 00:00:00 ~ 2025-01-01 23:59:59 결제 데이터



재처리가 가능한 Batch 시나리오

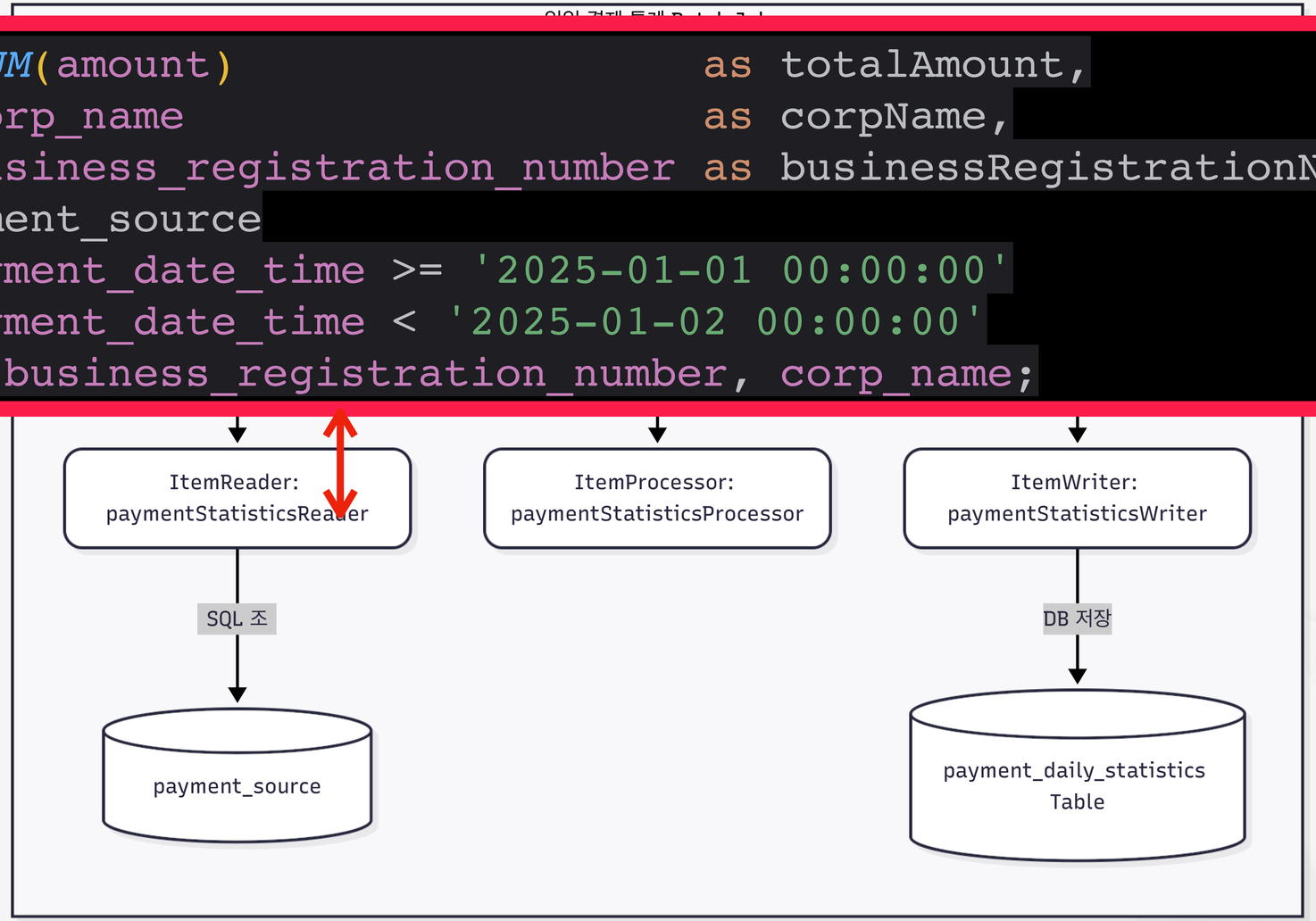


재처리가 가능한 Batch Flow



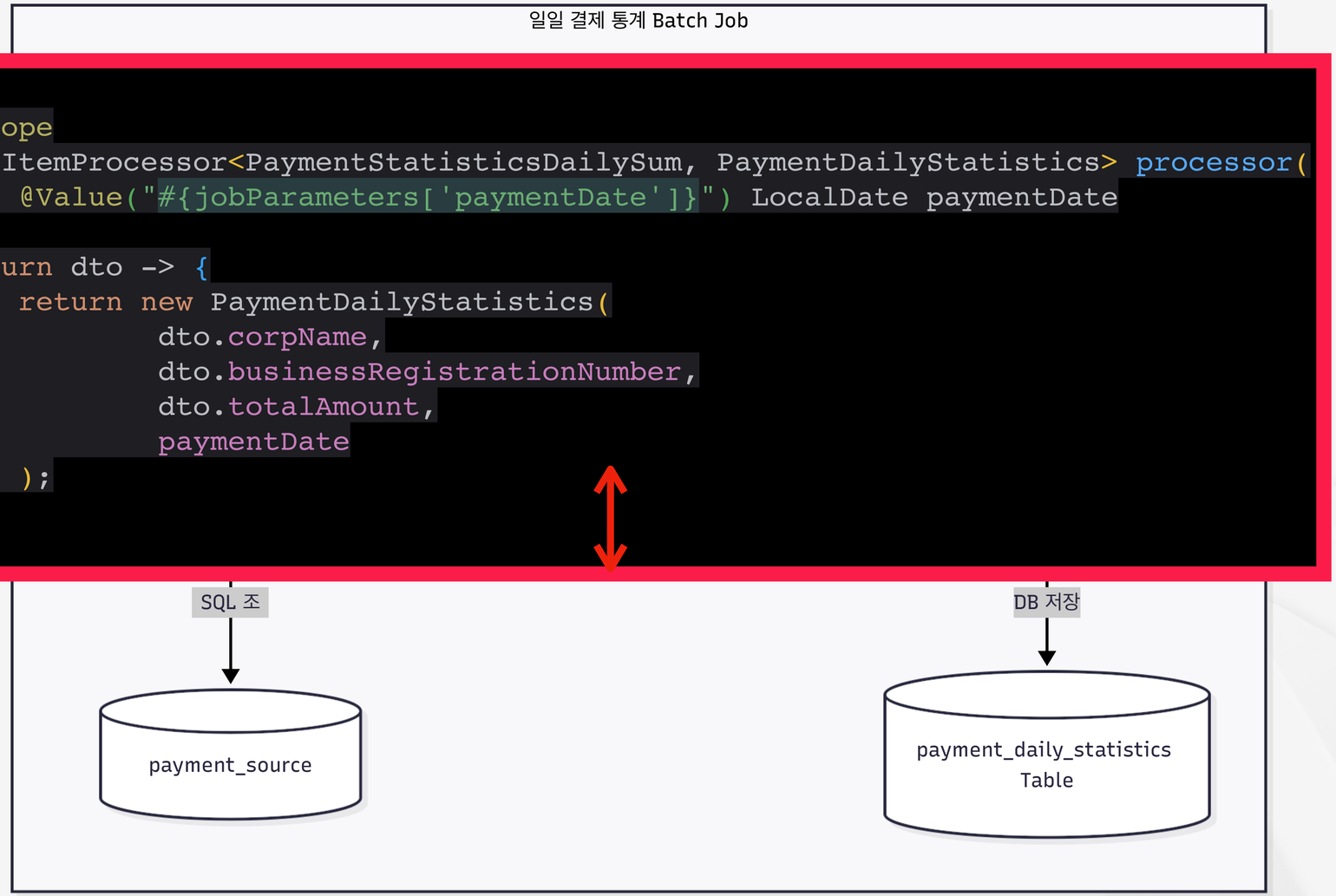
재처리가 가능한 Batch Flow

```sql
SELECT SUM(amount)                        as totalAmount,
       corp_name                          as corpName,
       business_registration_number as businessRegistrationNumber
FROM payment_source
WHERE payment_date_time >= '2025-01-01 00:00:00'
  AND payment_date_time < '2025-01-02 00:00:00'
GROUP BY business_registration_number, corp_name;
```



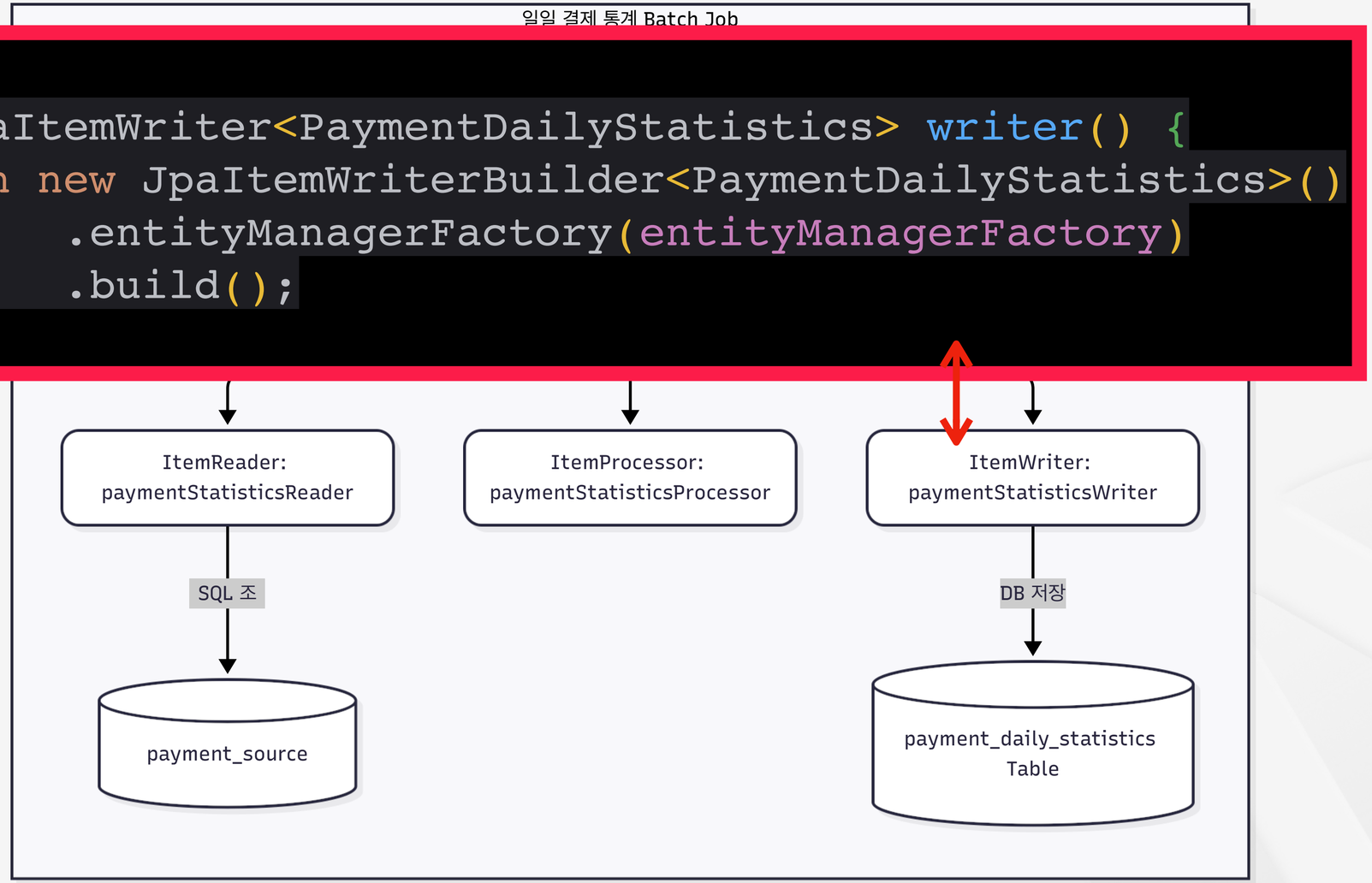
재처리가 가능한 Batch Flow

```java
@Bean
@StepScope
public ItemProcessor<PaymentStatisticsDailySum, PaymentDailyStatistics> processor(
        @Value("#{jobParameters['paymentDate']}") LocalDate paymentDate
) {
    return dto -> {
        return new PaymentDailyStatistics(
                dto.corpName,
                dto.businessRegistrationNumber,
                dto.totalAmount,
                paymentDate
        );
    };
}
```

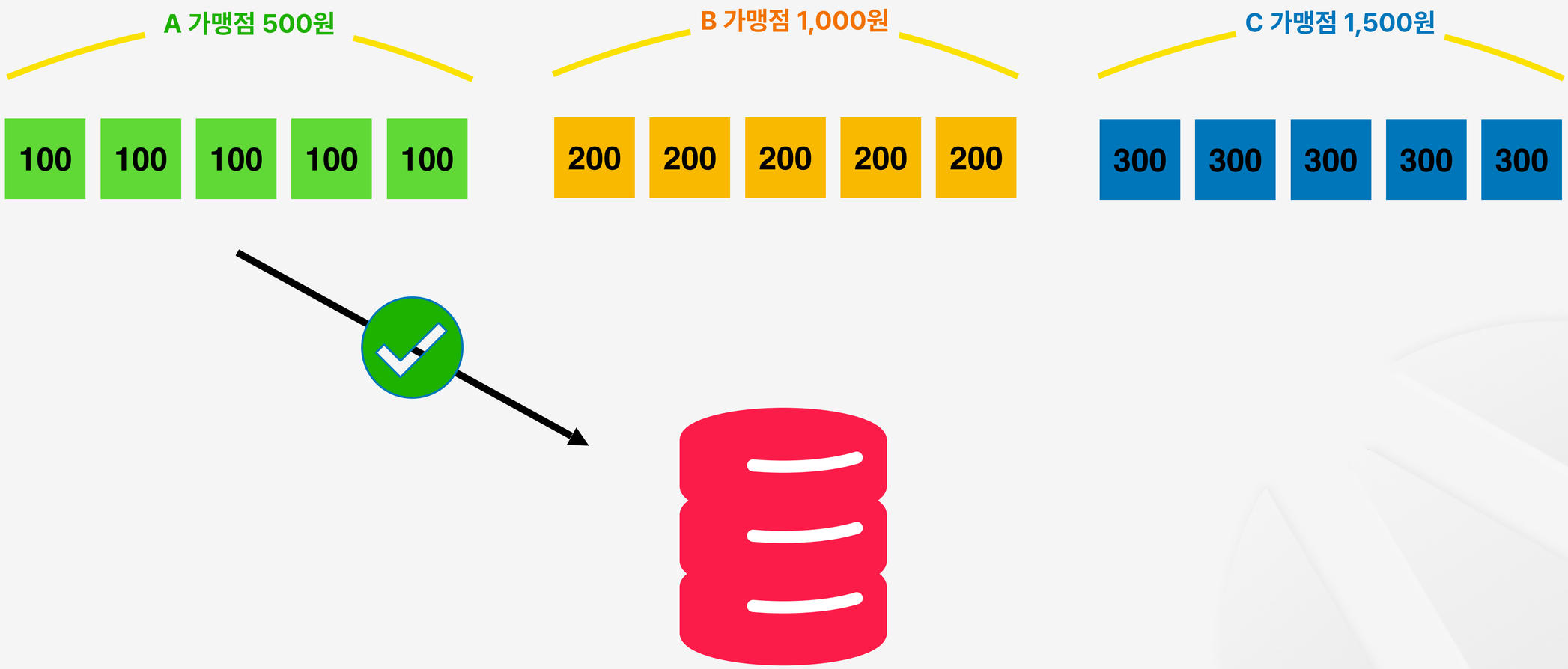


재처리가 가능한 Batch Flow

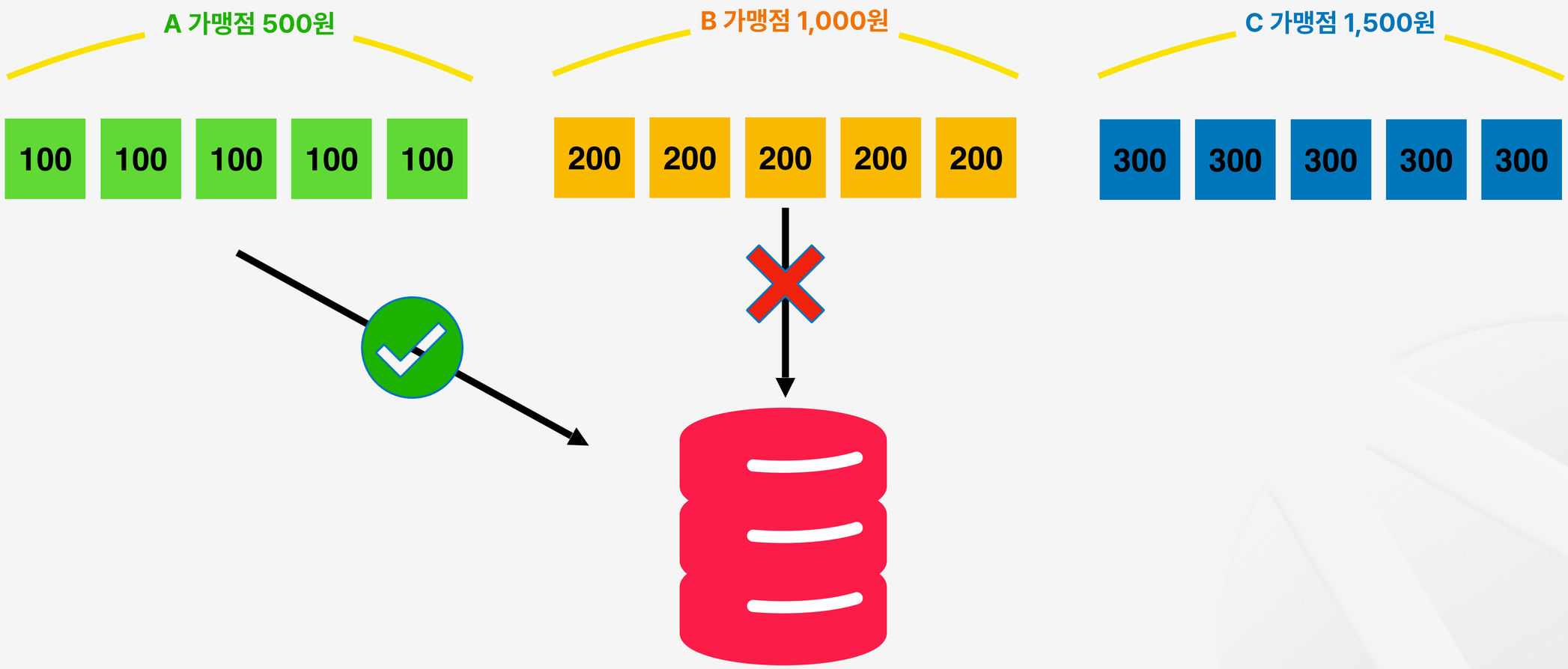
```java
@Bean
public JpaItemWriter<PaymentDailyStatistics> writer() {
    return new JpaItemWriterBuilder<PaymentDailyStatistics>()
            .entityManagerFactory(entityManagerFactory)
            .build();
}
```



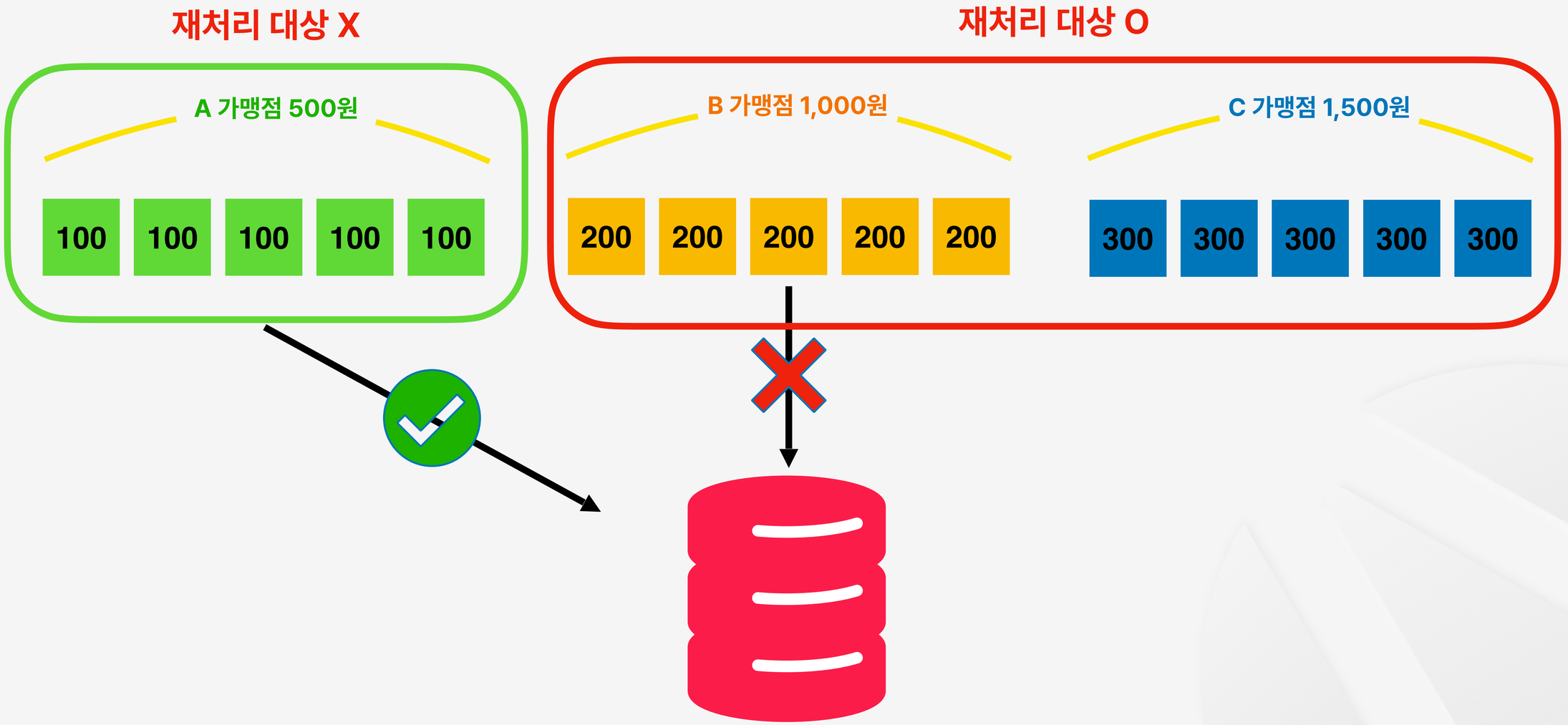
일 통계통계 Job의 문제점: 재처리



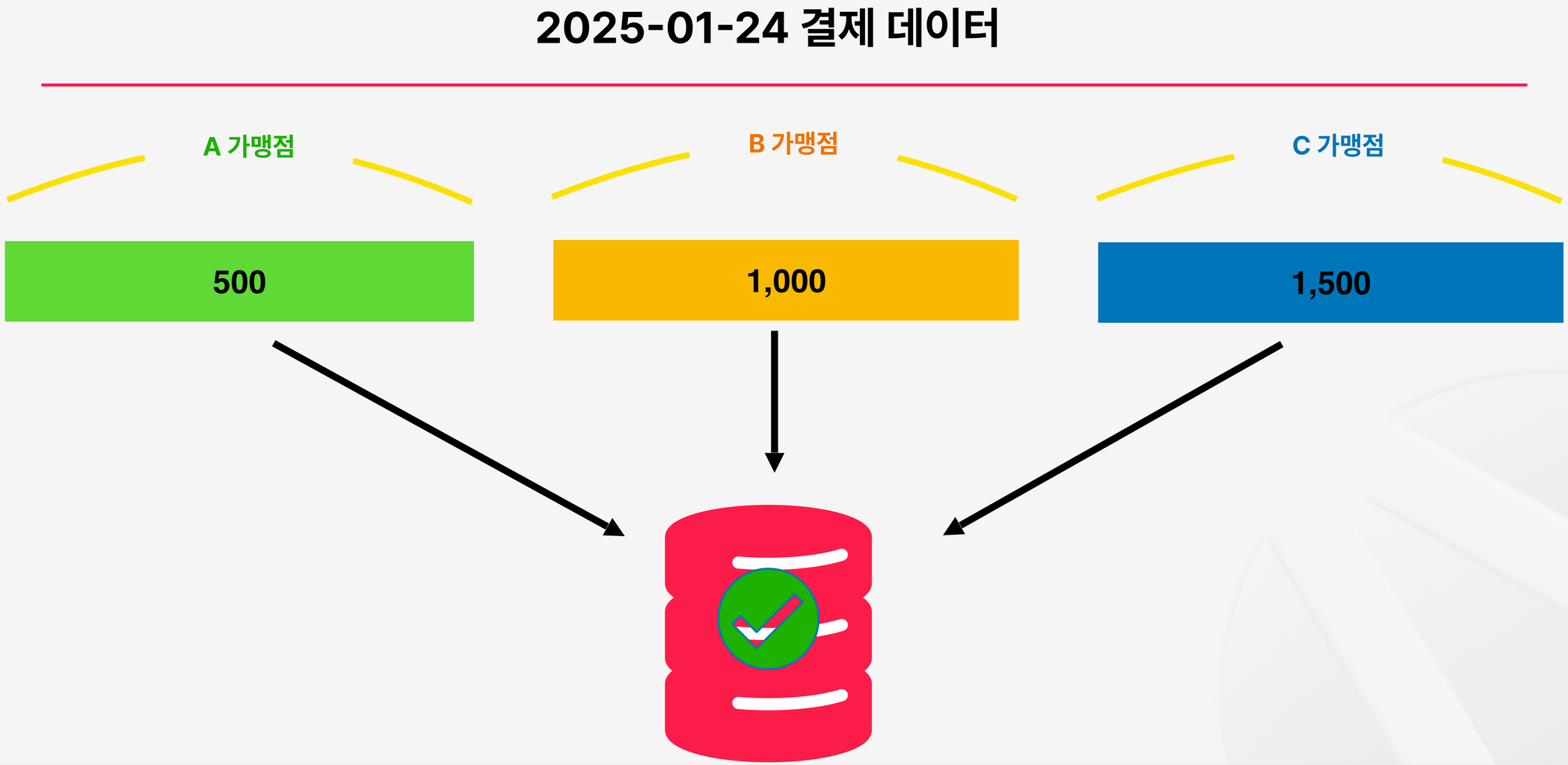
일 통계통계 Job의 문제점: 재처리



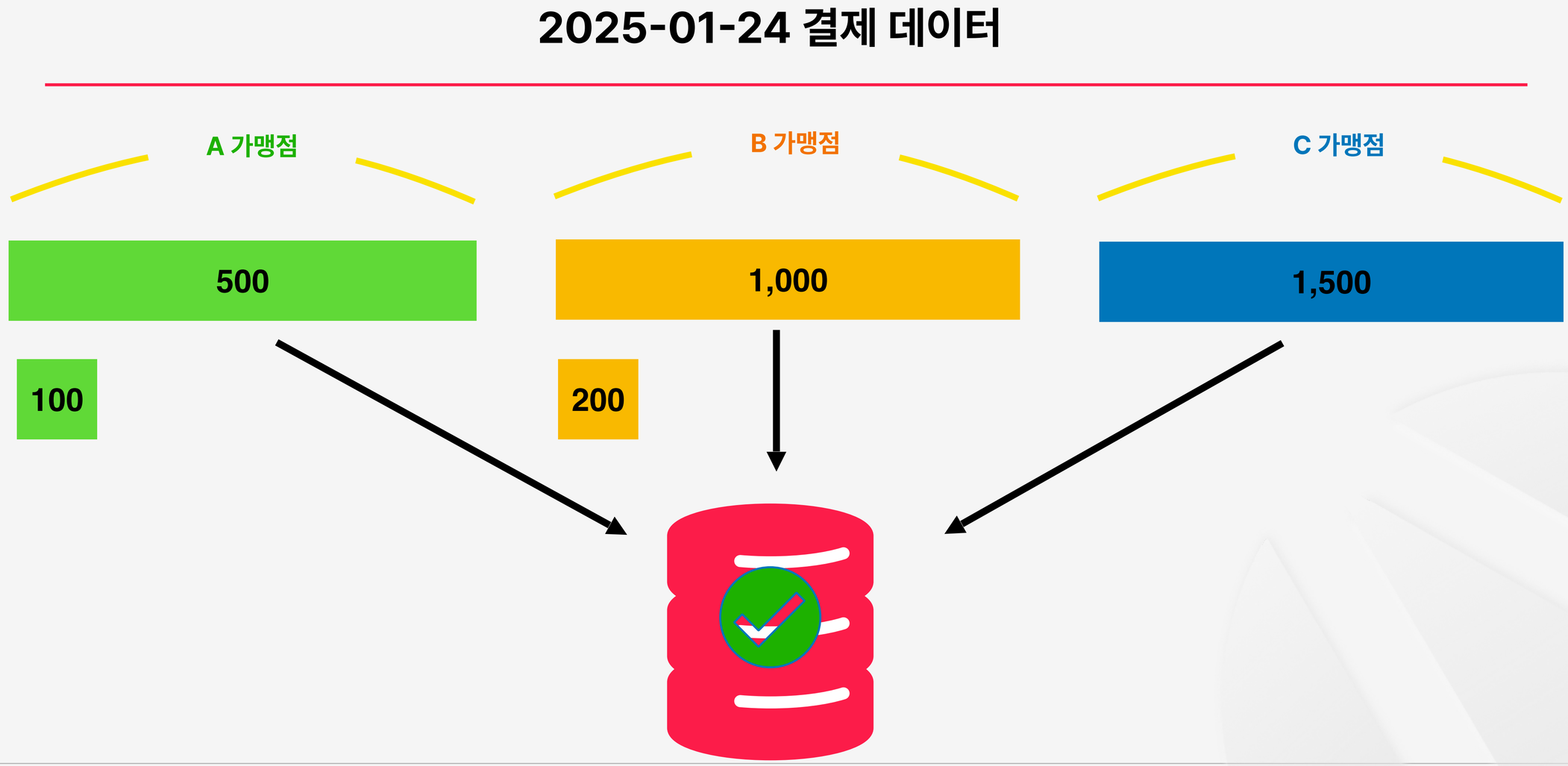
일 통계통계 Job의 문제점: 재처리



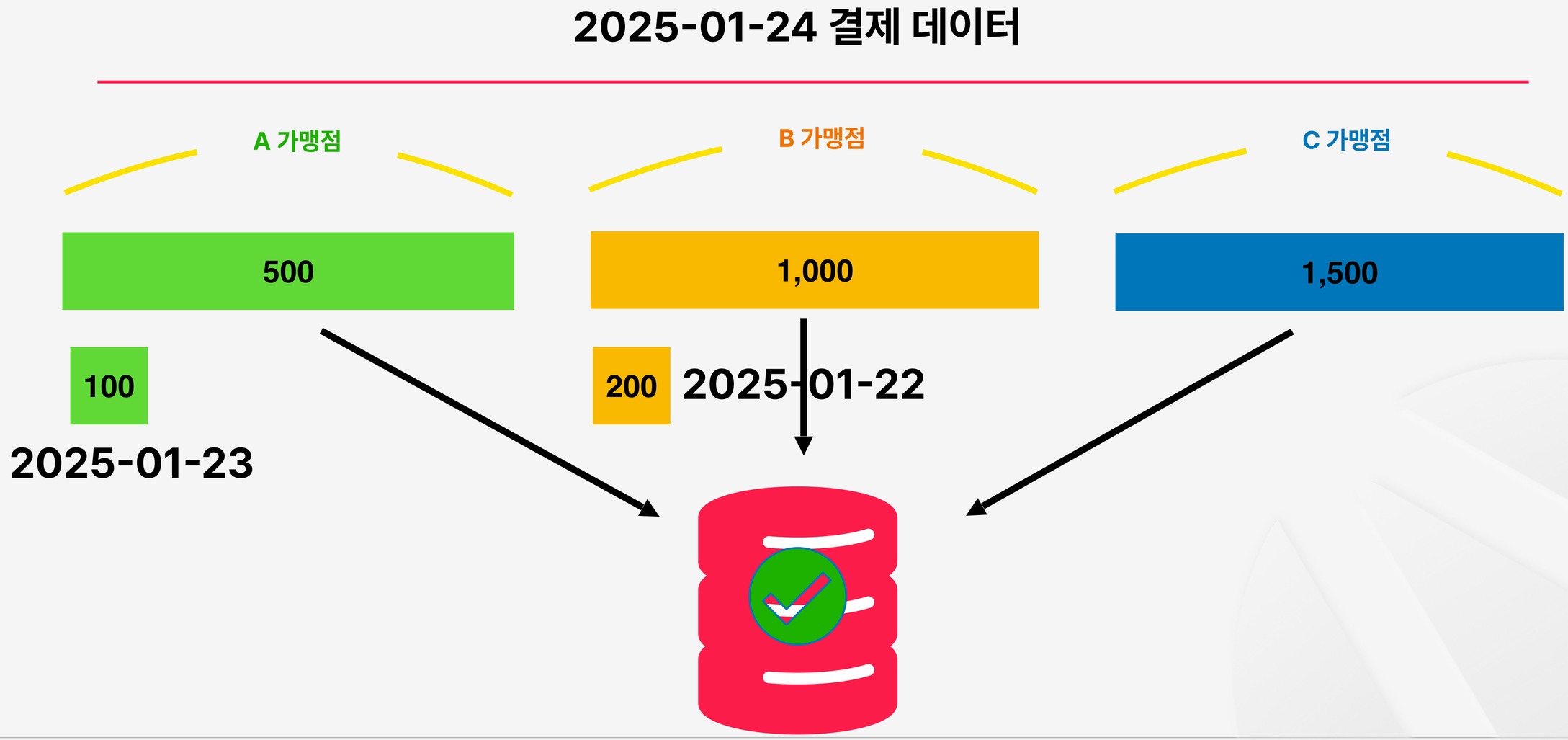
일 통계통계 Job의 문제점: 추가 데이터



일 통계통계 Job의 문제점: 추가 데이터



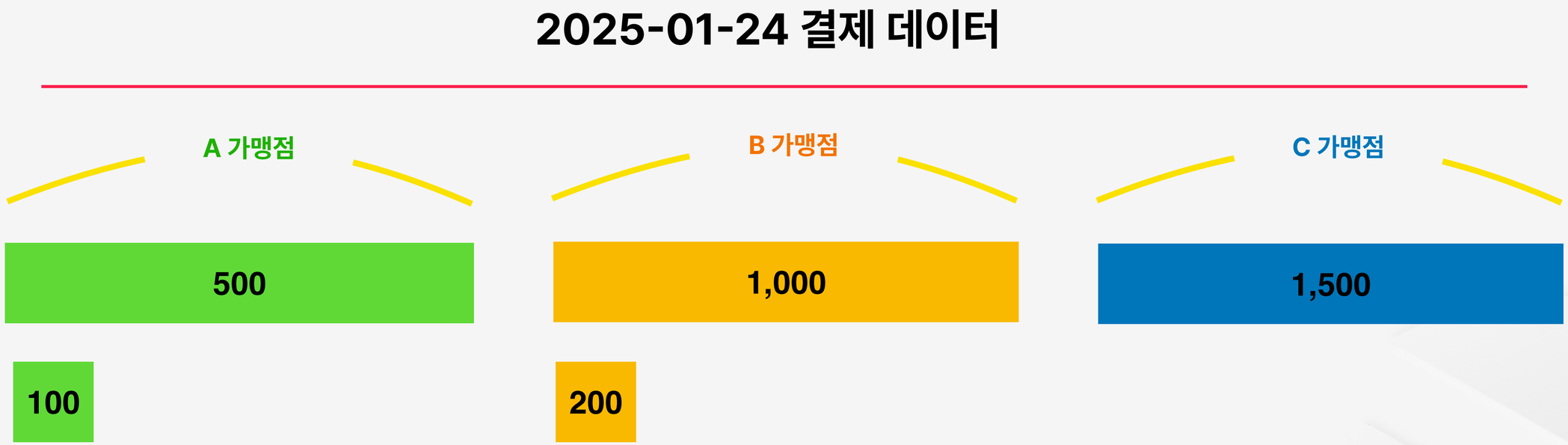
일 통계통계 Job의 문제점: 추가 데이터



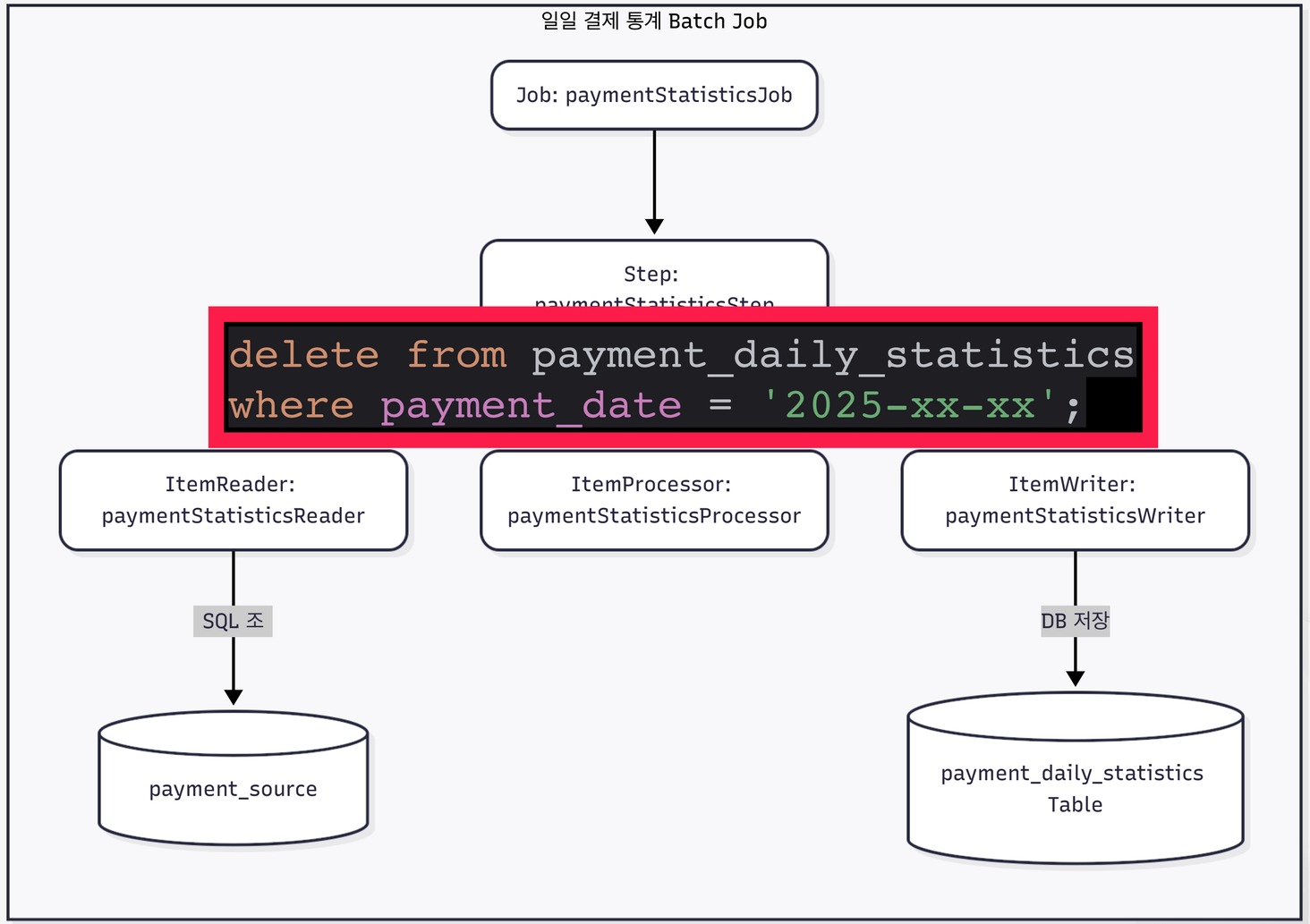
일 통계통계 Job의 문제점: 가장 쉬운 해결 방법



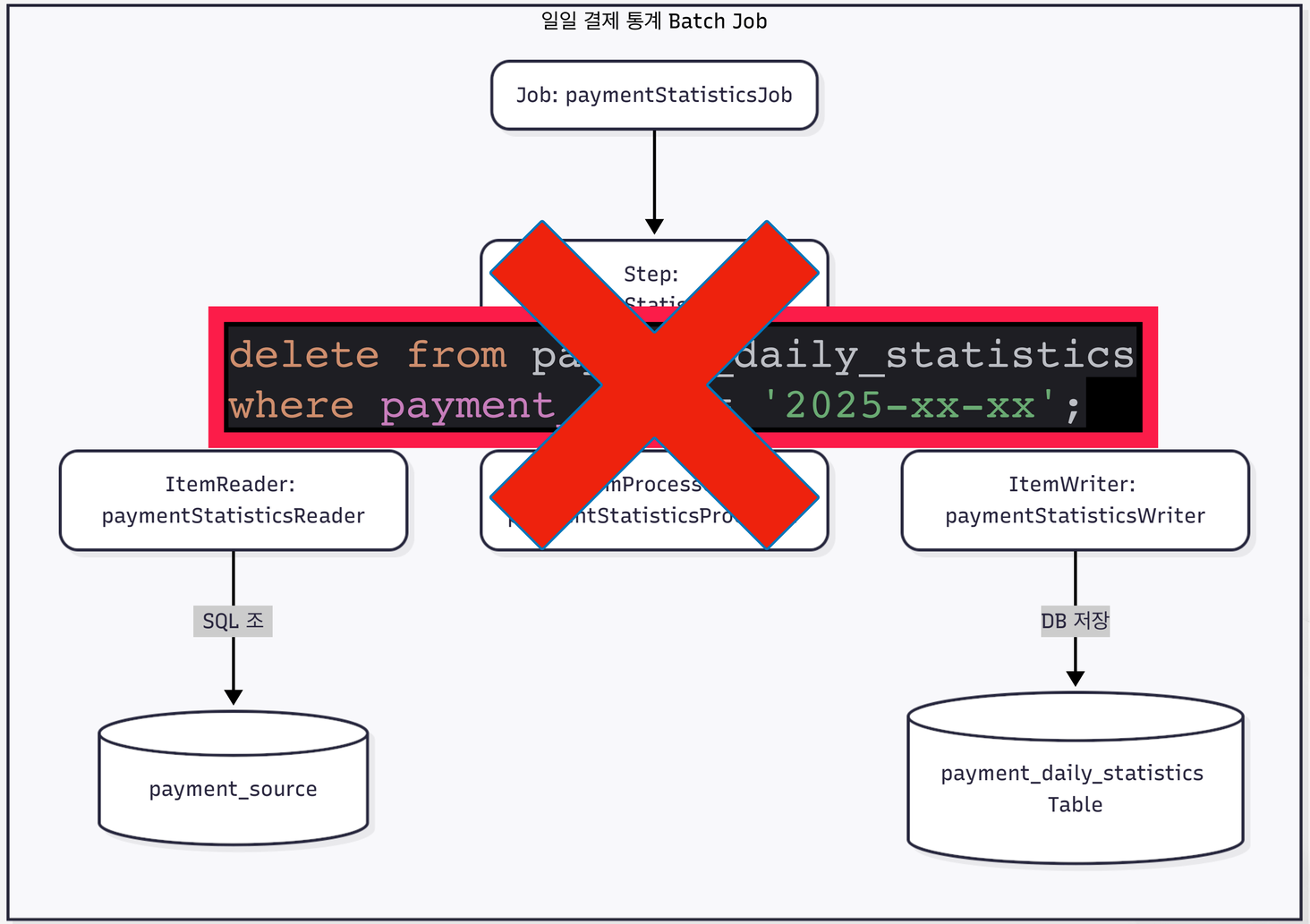
일 통계통계 Job의 문제점: 가장 쉬운 해결 방법



데이터 삭제 이후 일 통계통계 Job 재처리



데이터 삭제 이후 일 통계통계 Job 재처리



ONLY_FULL_GROUP_BY 란?

- 정의
  - MySQL 5.7.5부터 기본으로 활성화된 엄격한 SQL 모드입니다.
  - GROUP BY를 사용하는 쿼리의 모호성을 없애고 데이터의 일관성을 보장하기 위함입니다.
- 규칙
  - SELECT 절에 있는 모든 컬럼은 다음 중 하나여야 합니다.
  - GROUP BY 절에 포함되어야 함.
  - SUM(), COUNT(), MIN(), MAX() 같은 집계 함수로 감싸져야 함.
  - 이 규칙을 위반하면 SQL 에러가 발생합니다.
- 예시
  - SELECT 사업자번호, 결제일시, SUM(금액) FROM 결제내역 GROUP BY 사업자번호;
  - 이 쿼리는 에러를 발생시킵니다. 결제일시는 GROUP BY에 포함되지 않았고, 집계 함수로 감싸지도 않았기 때문입니다.
  - MySQL은 어떤 결제일시를 보여줘야 할지 알 수 없습니다.

ONLY_FULL_GROUP_BY 란?

- 해결 방법
  - 쿼리 수정 (권장): 쿼리를 규칙에 맞게 수정합니다.
    - GROUP BY에 결제일시를 포함하거나, ANY_VALUE(결제일시) 함수를 사용합니다.
  - 설정 해제 (필요시): MySQL의 sql_mode에서 ONLY_FULL_GROUP_BY를 제거합니다.
    - SET GLOBAL sql_mode=(SELECT REPLACE(@@sql_mode,'ONLY_FULL_GROUP_BY',''));

재처리가 가능한 Batch

결제일시	금액	사업자번호
2025-01-01 00:01:02	100	사업자1
2025-01-01 23:01:02	300	사업자1
2025-01-01 00:01:02	1,000	사업자2

결제일	금액	사업자번호
2025-01-01	400	사업자1
2025-01-01	1,000	사업자2

재처리가 가능한 Batch - Update Case

결제일시	금액	사업자번호
2025-01-01 00:01:02	100	사업자1
2025-01-01 23:01:02	300	사업자1
2025-01-01 23:30:02	**100**	**사업자1**
2025-01-01 00:01:02	1,000	사업자2

결제일	금액	사업자번호
2025-01-01	400	사업자1
2025-01-01	1,000	사업자2

재처리가 가능한 Batch - Update Case

결제일시	금액	사업자번호
2025-01-01 00:01:02	100	사업자1
2025-01-01 23:01:02	300	사업자1
2025-01-01 23:30:02	**100**	**사업자1**
2025-01-01 00:01:02	1,000	사업자2

결제일	금액	사업자번호
2025-01-01	400	사업자1
2025-01-01	1,000	사업자2

재처리가 가능한 Batch - Insert Case

결제일시	금액	사업자번호
2025-01-01 00:01:02	100	사업자1
2025-01-01 23:01:02	300	사업자1
2025-01-01 23:30:02	100	사업자1
2025-01-01 00:01:02	1,000	사업자2
2025-01-01 02:01:02	**2,000**	**사업자3**

결제일	금액	사업자번호
2025-01-01	500	사업자1
2025-01-01	1,000	사업자2
2025-01-01	**2,000(insert)**	**사업자3**

재처리가 가능한 Batch - 변경 없는 Case

결제일시	금액	사업자번호
2025-01-01 00:01:02	100	사업자1
2025-01-01 23:01:02	300	사업자1
2025-01-01 23:30:02	100	사업자1
2025-01-01 00:01:02	1,000	사업자2
2025-01-01 02:01:02	2,000	사업자3
2025-01-01 02:01:22	**-2,000**	**사업자3**
2025-01-01 02:02:02	**2,000**	**사업자3**

결제일	금액	사업자번호
2025-01-01	500	사업자1
2025-01-01	1,000	사업자2
2025-01-01	2,000	사업자3

데이터 삭제 이후 재처리 Job

```java
@ConfigurationProperties(prefix = "args")
public class ArgumentProperties {

    /**
     * 결제 일자
     */
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    private LocalDate paymentDate;

    /**
     * 데이터 삭제 여부
     */
    private Boolean clearExistingData;
}
```

데이터 삭제 이후 재처리 Job

```
public class ClearExistingDataJobListener implements JobExecutionListener {

    private final PaymentDailyStatisticsRepository paymentDailyStatisticsRepository;
    private final ArgumentProperties properties;

    @Override
    public void beforeJob(JobExecution jobExecution) {
        if (properties.getClearExistingData()){
            log.info("paymentDate='{}'에 해당하는 기존 결제 통계 데이터 삭제를 시작합니다.",
                properties.getPaymentDate());
            paymentDailyStatisticsRepository.deleteByPaymentDate(properties.getPaymentDate());
            log.info("기존 결제 통계 데이터 삭제를 완료했습니다.");
        }
    }

    @Override
    public void afterJob(JobExecution jobExecution) {}
}
```

데이터 삭제 이후 재처리 Job

```
public class ClearExistingDataJobListener implements JobExecutionListener {

    private final PaymentDailyStatisticsRepository paymentDailyStatisticsRepository;
    private final ArgumentProperties properties;

    @Modifying
    @Query("delete from PaymentDailyStatistics p where p.paymentDate = :paymentDate")
    void deleteByPaymentDate(@Param("paymentDate") LocalDate paymentDate);
                properties.getPaymentDate());
            paymentDailyStatisticsRepository.deleteByPaymentDate(properties.getPaymentDate());
            log.info("기존 결제 통계 데이터 삭제를 완료했습니다.");
        }
    }

    @Override
    public void afterJob(JobExecution jobExecution) {}
}
```

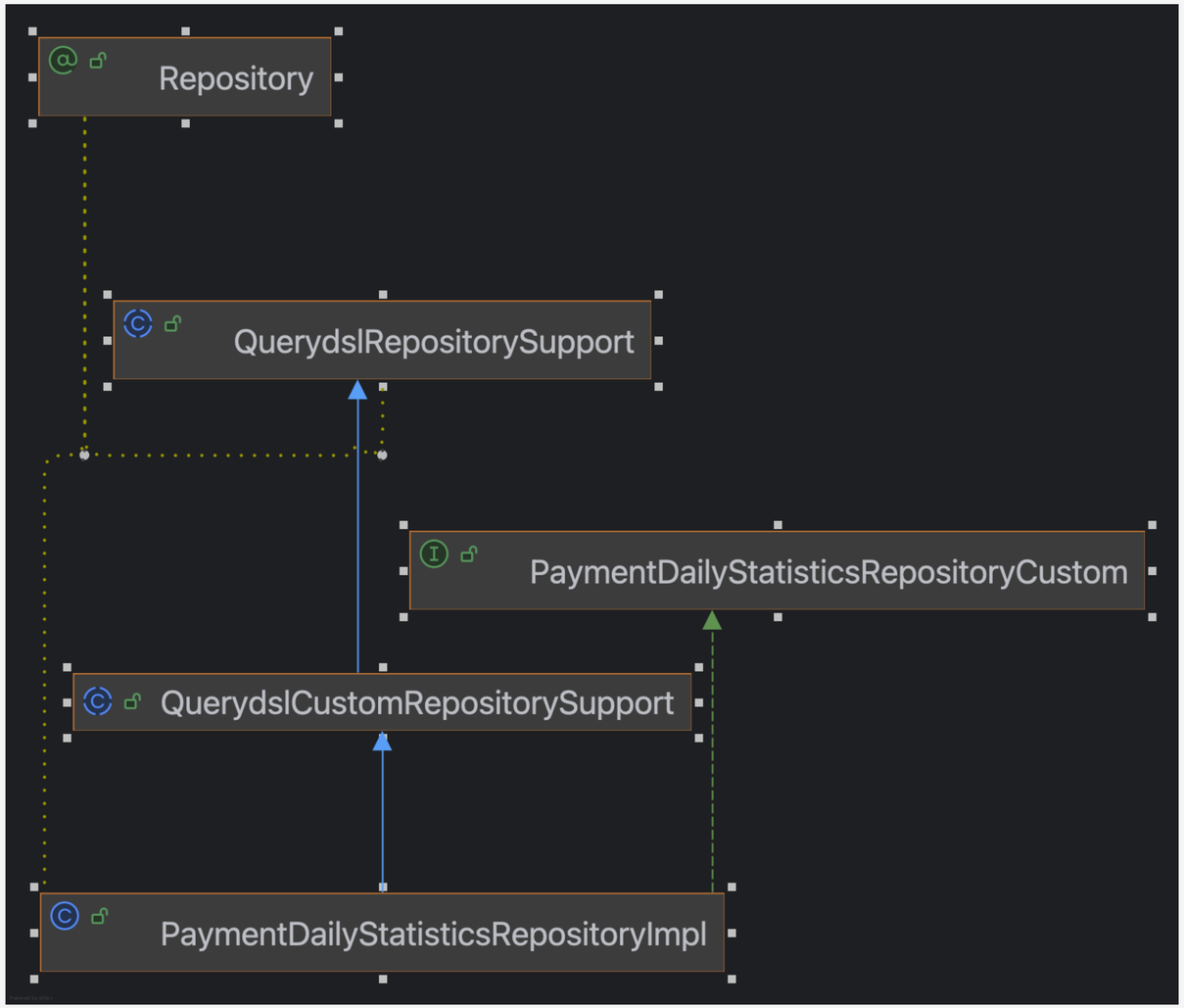
자동 재처리 Batch Querydsl

```
// QueryDSL-JPA (읽기/조회용으로 필요할 수 있음)
implementation 'com.querydsl:querydsl-jpa:5.1.0:jakarta'
annotationProcessor 'com.querydsl:querydsl-apt:5.1.0:jakarta'
annotationProcessor "jakarta.annotation:jakarta.annotation-api"
annotationProcessor "jakarta.persistence:jakarta.persistence-api"

// QueryDSL-SQL (QuerydslBatchItemWriter에서 사용)
implementation 'com.querydsl:querydsl-sql:5.1.0'

// QueryDSL-SQL 코드 생성을 위한 의존성 (Annotation Processor가 아님)
// 이 의존성은 별도의 Gradle Task에서 사용됩니다.
implementation 'com.querydsl:querydsl-sql-codegen:5.1.0'
```

자동 재처리 Batch Querydsl



자동 재처리 Batch Querydsl

```java
@Repository
public class PaymentDailyStatisticsRepositoryImpl extends
 QuerydslCustomRepositorySupport implements PaymentDailyStatisticsRepositoryCustom {

    public PaymentDailyStatisticsRepositoryImpl() {
        super(PaymentDailyStatistics.class);
    }

    @Override
    @Transactional
    public long deleteByPaymentDate(LocalDate paymentDate) {
        return delete(paymentDailyStatistics)
                .where(paymentDailyStatistics.paymentDate.eq(paymentDate))
                .execute();
    }
}
```

자동 재처리 Batch 정리

- **멱등성(Idempotency)**
  - 동일한 작업을 여러 번 수행해도 최종 결과는 항상 같도록 보장하는 것.
  - 데이터 누락, 오류, 재처리 요청 등 어떠한 상황에서도 **'일관된 결과'**를 만들어내는 것이 목표.
- **고유 키 기반 처리**
  - **사업자 번호 + 결제 일자**라는 유니크한 조합을 기준으로 로직을 설계.
  - 이는 '**일일 집계**'라는 작업의 본질적인 고유성을 나타내며, 변화하지 않는 기준이 됨.
- 시나리오별 대응 로직
  - INSERT 케이스: 해당 고유 키 조합의 데이터가 최종 집계 테이블에 없을 경우, 신규 레코드 INSERT 실행.
  - UPDATE 케이스: 해당 고유 키 데이터는 있지만, 원천 데이터를 다시 합산한 최종 금액이 다를 경우, 기존 레코드 UPDATE 실행.
  - 변경 없는 케이스: 원천 데이터는 변경되었더라도, 최종 집계 금액이 동일할 경우, 별도의 DB 연산 없이 그대로 둠.
- 결과적으로
  - 이와 같은 로직을 통해 우리는 어떤 시점에 배치 잡을 재실행하더라도, 항상 가장 정확한 최종 상태의 데이터를 얻을 수 있습니다.
  - 이는 곧 시스템의 신뢰성을 획기적으로 향상시키는 방법입니다.

재처리가 가능한 Batch 문제

결제일시	금액	사업자번호
2025-01-05 00:01:02	100	사업자1
2025-01-05 23:01:02	300	사업자1
2025-01-05 00:01:02	1,000	사업자2

결제일	금액	사업자번호
2025-01-05	400	사업자1
2025-01-05	1,000	사업자2

재처리가 가능한 Batch 문제

결제일시	금액	사업자번호
2025-01-05 00:01:02	100	사업자1
2025-01-05 23:01:02	300	사업자1
2025-01-05 00:01:02	1,000	사업자2
2025-01-04 00:01:02	**500**	**사업자1**
2025-01-03 00:01:02	**500**	**사업자2**
2025-01-03 00:01:02	**500**	**사업자3**

결제일	금액	사업자번호
2025-01-05	400	사업자1
2025-01-05	1,000	사업자2

재처리가 가능한 Batch 문제

결제일시	금액	사업자번호
2025-01-05 00:01:02	100	사업자1
2025-01-05 2		
2025-01-05 0		
2025-01-04 0		
2025-01-03 0		
2025-01-03 00:01:02	500	사업자3

```sql
SELECT SUM(amount)                      as totalAmount,
       corp_name                        as corpName,
       business_registration_number as businessRegistrationNumber
FROM payment_source
WHERE payment_date_time >= '2025-01-05 00:00:00'
  AND payment_date_time < ‘2025-01-06 00:00:00'
GROUP BY business_registration_number, corp_name;
```

결제일	금액	사업자번호
		사업자1
		사업자2

재처리가 가능한 Batch 문제 해결

결제일시	금액	사업자번호	updated_at
2025-01-05 00:01:02	100	사업자1	2025-01-05 19:01:02
2025-01-05 23:01:02	300	사업자1	2025-01-05 19:01:02
2025-01-05 00:01:02	1,000	사업자2	2025-01-05 19:01:02
2025-01-04 00:01:02	**500**	**사업자1**	**2025-01-05 19:01:02**
2025-01-03 00:01:02	**500**	**사업자2**	**2025-01-05 19:01:02**
2025-01-03 00:01:02	**500**	**사업자3**	**2025-01-05 19:01:02**

재처리가 가능한 Batch 문제 해결

결제일시	금액	사업자번호	updated_at

```
selectFrom(paymentSource)
        .where(paymentSource.updatedAt.between(
                LocalDate.now().atStartOfDay(),
                LocalDateTime.of(LocalDate.now(), LocalTime.MAX)
        ))
        .fetch()
        .stream()
        .map(it -> it.getPaymentDateTime().toLocalDate())
        .collect(Collectors.toSet());
```

재처리가 가능한 Batch 문제 해결

결제일시	금액	사업자번호	updated_at

```
selectFrom(paymentSource)
        .where(paymentSource.updatedAt.between(
                LocalDate.now().atStartOfDay()
                2025-01-05, 2025-01-04, 2025-01-03
        .fetch()
        .stream()
        .map(it -> it.getPaymentDateTime().toLocalDate())
        .collect(Collectors.toSet());
```

재처리가 가능한 Batch 문제 해결

결제일시	금액	사업자번호	updated_at

```sql
SELECT SUM(amount)                    as totalAmount,
       corp_name                      as corpName,
       business_registration_number   as businessRegistrationNumber,
       DATE(payment_date_time)        as paymentDate
FROM payment_source
WHERE (payment_date_time >= '2025-01-05 00:00:00' AND payment_date_time < '2025-01-06 00:00:00')
   OR (payment_date_time >= '2025-01-04 00:00:00' AND payment_date_time < '2025-01-05 00:00:00')
   OR (payment_date_time >= '2025-01-03 00:00:00' AND payment_date_time < '2025-01-04 00:00:00')
GROUP BY business_registration_number, corp_name, DATE(payment_date_time);
```

2025-01-03 00:01:02	500	사업자3	2025-01-05 19:01:02

재처리가 가능한 Batch 문제 해결

결제일시	금액	사업자번호
2025-01-05 00:01:02	100	사업자1
2025-01-05 23:01:02	300	사업자1
2025-01-05 00:01:02	1,000	사업자2
2025-01-04 00:01:02	**500**	**사업자1**
2025-01-03 00:01:02	**500**	**사업자2**
2025-01-03 00:01:02	**500**	**사업자3**

결제일	금액	사업자번호
2025-01-05	400	사업자1
2025-01-05	1,000	사업자2
2025-01-04	**500**	**사업자1**
2025-01-03	**500**	**사업자2**
2025-01-03	**500**	**사업자2**

자동 재처리 Batch 정리

- **핵심 원리: 멱등성(Idempotency)과 고유 키**
  - 멱등성(Idempotency)
    - 동일한 작업을 여러 번 수행해도 최종 결과는 항상 같도록 보장합니다.
    - 데이터 누락, 오류 등 어떤 상황에서도 **'일관된 결과'**를 만들어내는 것이 목표입니다.
- **고유 키 기반 처리**
  - **사업자 번호 + 결제 일자**라는 유니크한 조합을 기준으로 로직을 설계합니다.
  - 이 조합은 **일일 집계** 라는 작업의 본질적인 고유성을 나타내며, **변화하지 않는 기준**이 됩니다.
- 시나리오별 대응 로직
  - **INSERT 케이스**: 해당 고유 키 데이터가 최종 테이블에 **없을 경우**, 신규 레코드 `**INSERT**`실행
  - **UPDATE 케이스**: 해당 고유 키 데이터는 있지만, 재집계 금액이 **다를 경우**, 기존 레코드 `UPDATE`실행.
  - **변경 없는 케이스**: 원천 데이터가 변경되었더라도, 최종 금액이 **동일할 경우**, 별도의 DB 연산 없이 그대로 둠.

자동 재처리 Batch 정리

- 배치의 주기적 스케줄링:
  - 우리가 만든 배치는 보통 **매일 특정 시간**에 자동으로 실행되도록 스케줄링됩니다.
  - 이때, `updated_at` 컬럼을 활용하면 과거 데이터가 뒤늦게 유입되어도, 다음 번 배치가 실행될 때 자동으로 해당 변경분을 감지하고 재처리합니다.
- 타팀과의 협업에서 빛을 발하는 효과
  - 만약 다른 팀에서 데이터 누락분을 처리하고, 그 사실을 우리 팀에 별도로 공유하지 않더라도 걱정할 필요가 없습니다.
  - 해당 데이터의 **updated_at** 정보가 갱신되므로, 다음 배치 실행 시 시스템이 스스로 이 변경분을 감지하고 **자동으로 보정**해줍니다.
- 결과적으로.
  - 이러한 로직을 통해 우리는 어떤 시점에 배치 잡을 재실행하더라도, 항상 가장 정확한 최종 상태의 데이터를 얻을 수 있습니다
  - **배치가 돌면 돌수록 데이터 정합성이 자연스럽게 높아지는 강력한 효과**를 얻을 수 있으며, 이는 곧 시스템의 신뢰성을 획기적으로 향상시키는 가장 좋은 방법입니다.

Part 3 Spring Batch 실무에서의 배치

01

재처리가 가능한 Batch

- 멱등성 배치 어플리케이션 개발
- 데이터 누락이 케이스 대응

02

HTTP Page Item Reader

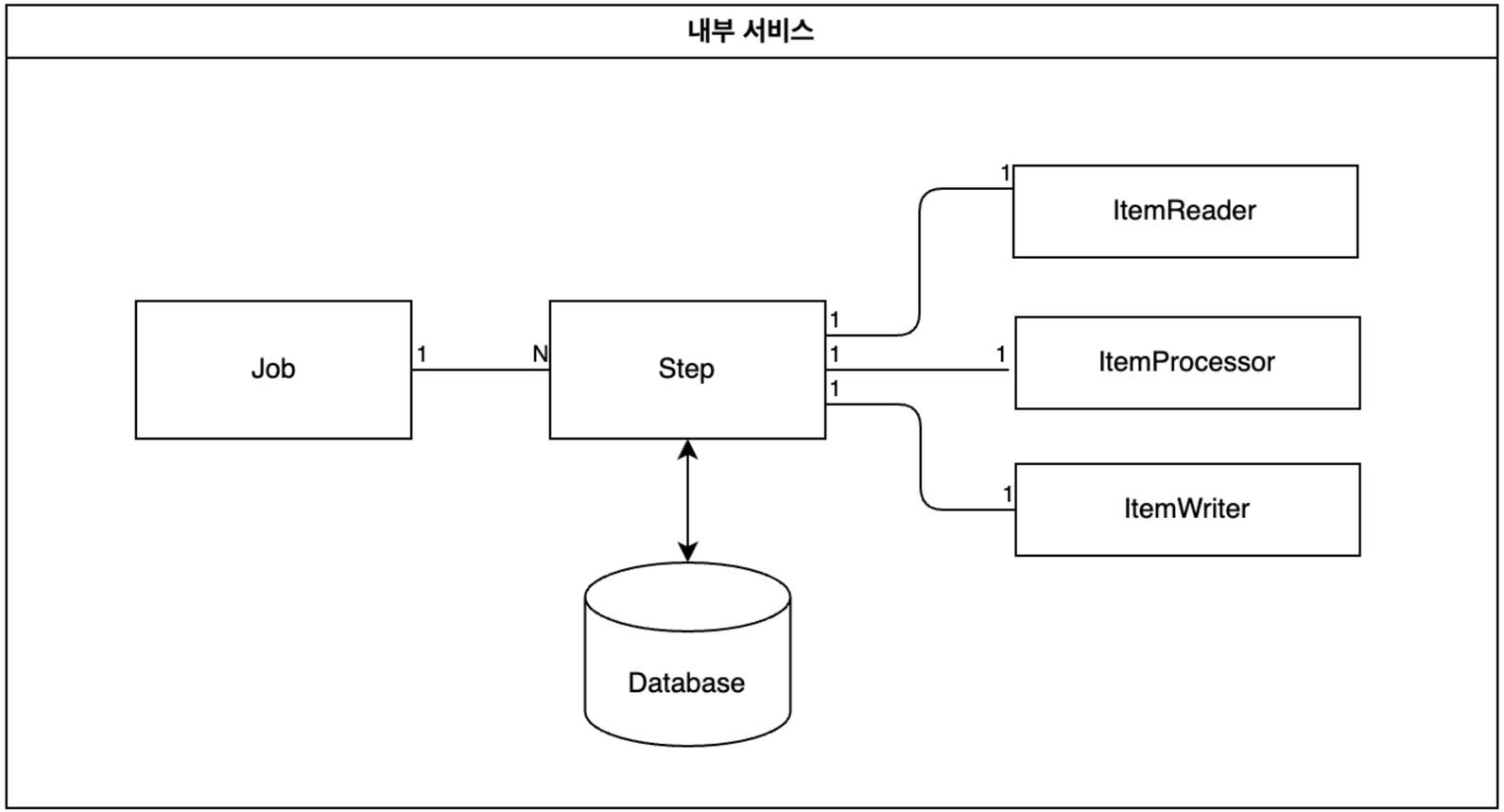
- 외부 데이터를 읽어 오는 방법
- 시스템의 강결합 해결

03

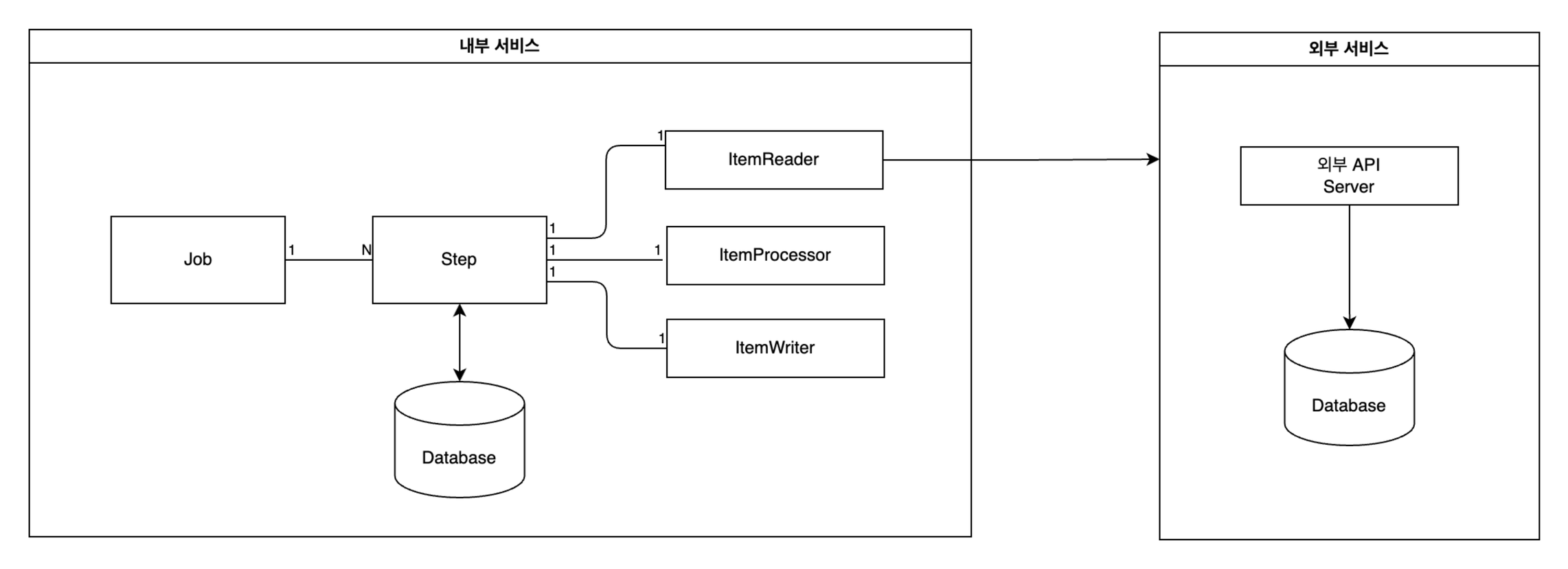
Batch Test 작성하는 노하우

- 다양한 테스트 케이스를 확보 하기 위한 방법
- 실무에서 의미 있는 테스트 코드를 작성하는 방법

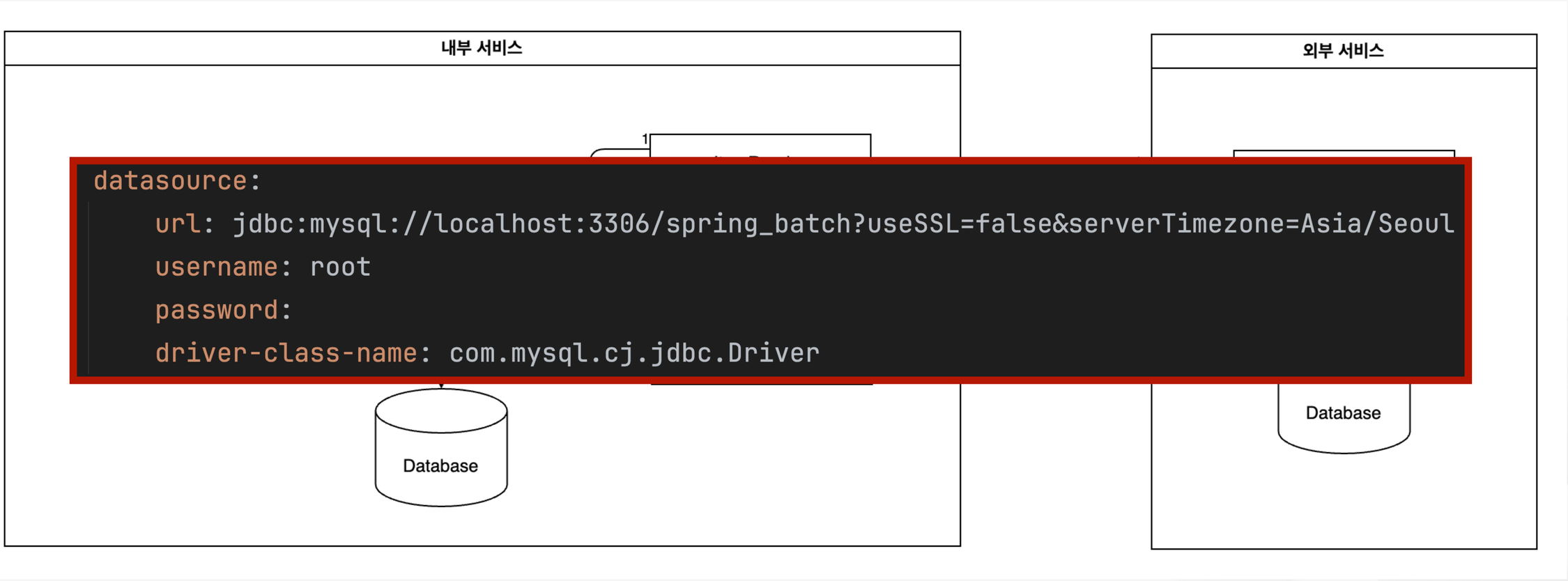
HTTP Page Item Reader - 외부 서버로 부터 데이터 읽어 오기



HTTP Page Item Reader - 외부 서버로 부터 데이터 읽어 오기

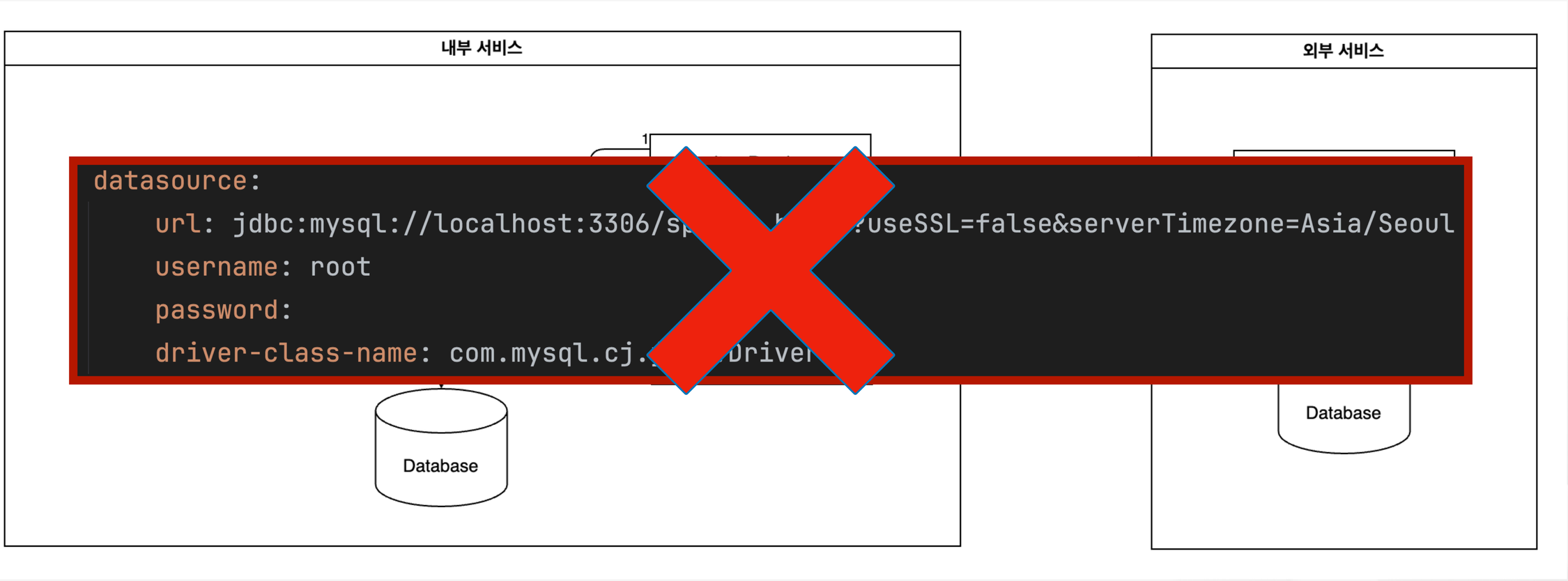


HTTP Page Item Reader - 외부 서버로 부터 데이터 읽어 오기



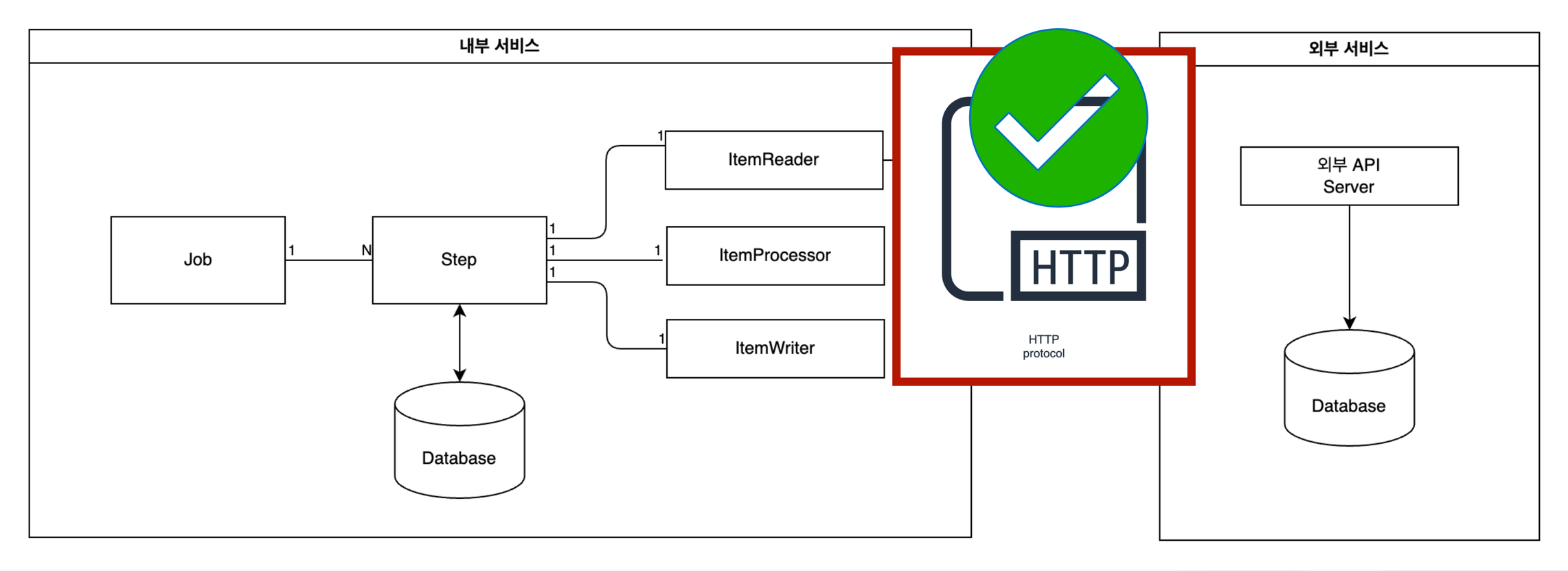

```yaml
datasource:
    url: jdbc:mysql://localhost:3306/spring_batch?useSSL=false&serverTimezone=Asia/Seoul
    username: root
    password:
    driver-class-name: com.mysql.cj.jdbc.Driver
```

HTTP Page Item Reader - 외부 서버로 부터 데이터 읽어 오기




```
datasource:
    url: jdbc:mysql://localhost:3306/sp      ?useSSL=false&serverTimezone=Asia/Seoul
    username: root
    password:
    driver-class-name: com.mysql.cj.      .Driver
```

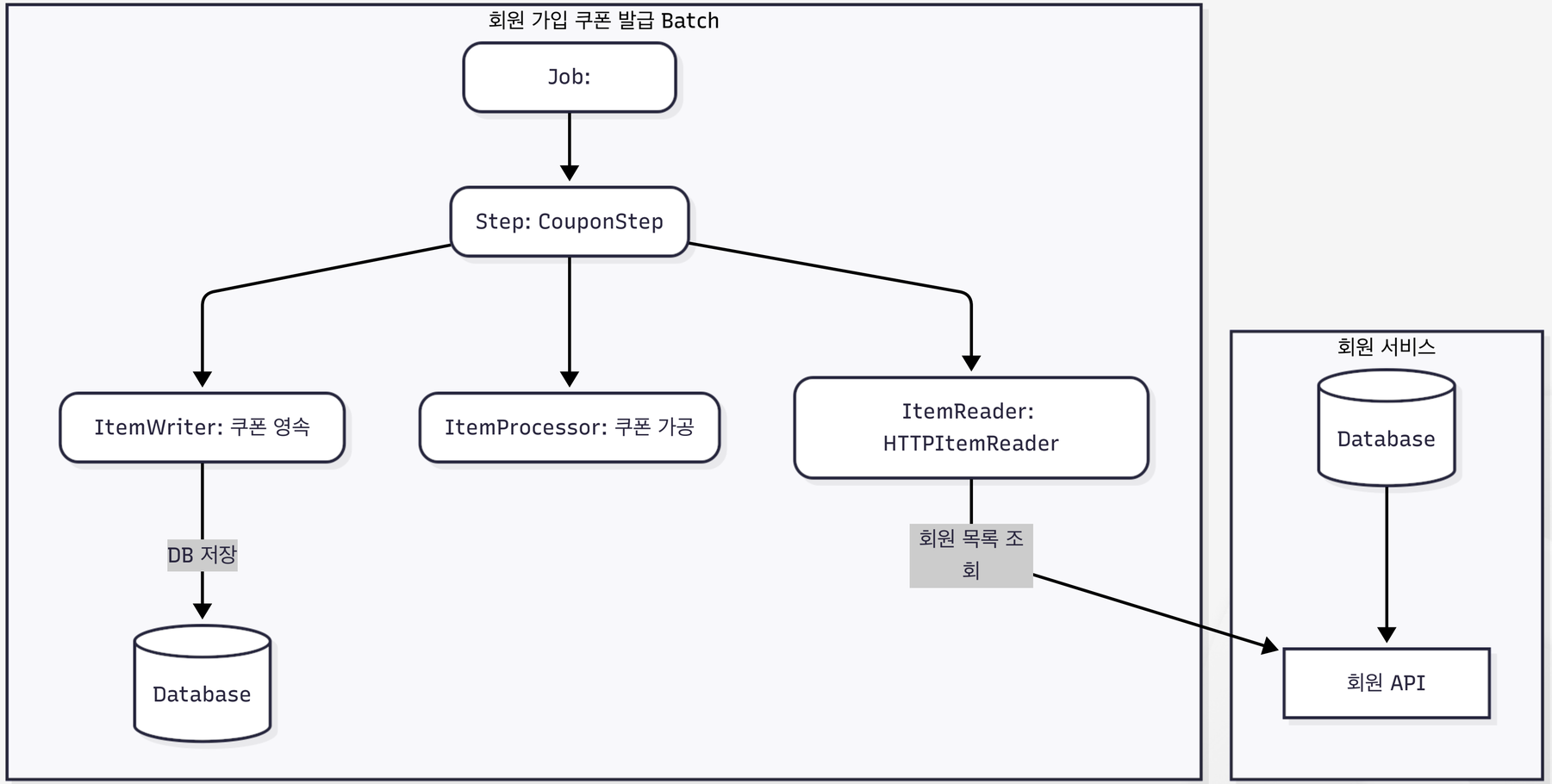
HTTP Page Item Reader - 외부 서버로 부터 데이터 읽어 오기



외부 데이터 접근 시 HTTP 통신을 권장 하는 이유

- 외부 시스템 의존성 감소
  - 접속 이슈 방지: 외부 데이터베이스 서버의 작업(증설, 버전 업데이트 등)이나 장애 발생 시, 직접 접속 방식은 우리 시스템에 직접적인 영향을 미칠 수 있습니다. HTTP 통신은 API 서버를 통해 간접적으로 데이터를 주고받기 때문에 이러한 의존성을 낮춥니다.
  - 접속 정보 관리: 외부 데이터베이스의 접속 정보 변경 시, 우리 시스템의 설정을 변경해야 하는 번거로움이 있습니다. API 서버를 이용하면 접속 정보 관리를 외부 팀에서 담당하게 되어 편리합니다.
- 데이터 무결성 및 보안 강화
  - 의도치 않은 데이터 변경 방지: 직접 접근 시 JPA의 더티 체킹 등으로 인해 의도하지 않은 쓰기 작업이 발생할 수 있습니다. HTTP 통신은 명확히 정의된 API를 통해 데이터 접근 권한을 제어하므로 데이터 무결성을 유지하고 불필요한 쓰기 작업을 방지할 수 있습니다.
  - 접근 권한 관리 용이: 외부 데이터베이스에 직접적인 쓰기 권한을 부여하는 것은 보안상 위험할 수 있습니다. API 서버를 통해 필요한 데이터만 제공하고 쓰기 작업은 API를 통해서만 이루어지도록 통제할 수 있습니다.
- 변경 사항에 대한 유연한 대응
  - 엔티티 변경 영향 최소화: 외부 데이터베이스의 엔티티 구조가 변경될 경우, 직접 접근하는 시스템은 수정이 불가피합니다. API 서버를 통해 데이터를 주고받으면, 외부 팀에서 API 변경을 통해 영향 범위를 최소화하고 호환성을 유지할 수 있습니다.
  - API 중심의 협업: API를 통해 데이터가 제공되면, 데이터 제공 팀은 API 명세에 따라 변경 사항을 관리하고, 데이터 소비 팀은 명세를 기반으로 시스템을 개발하므로 협업이 원활해집니다.

회원 가입 쿠폰 발급 Job Flow



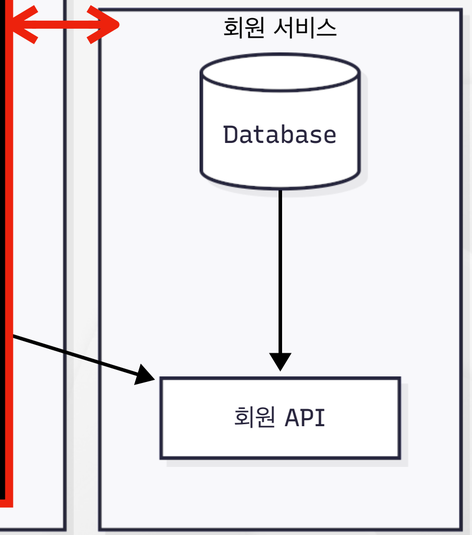
회원 가입 쿠폰 발급 Job Flow

```java
public class Member {

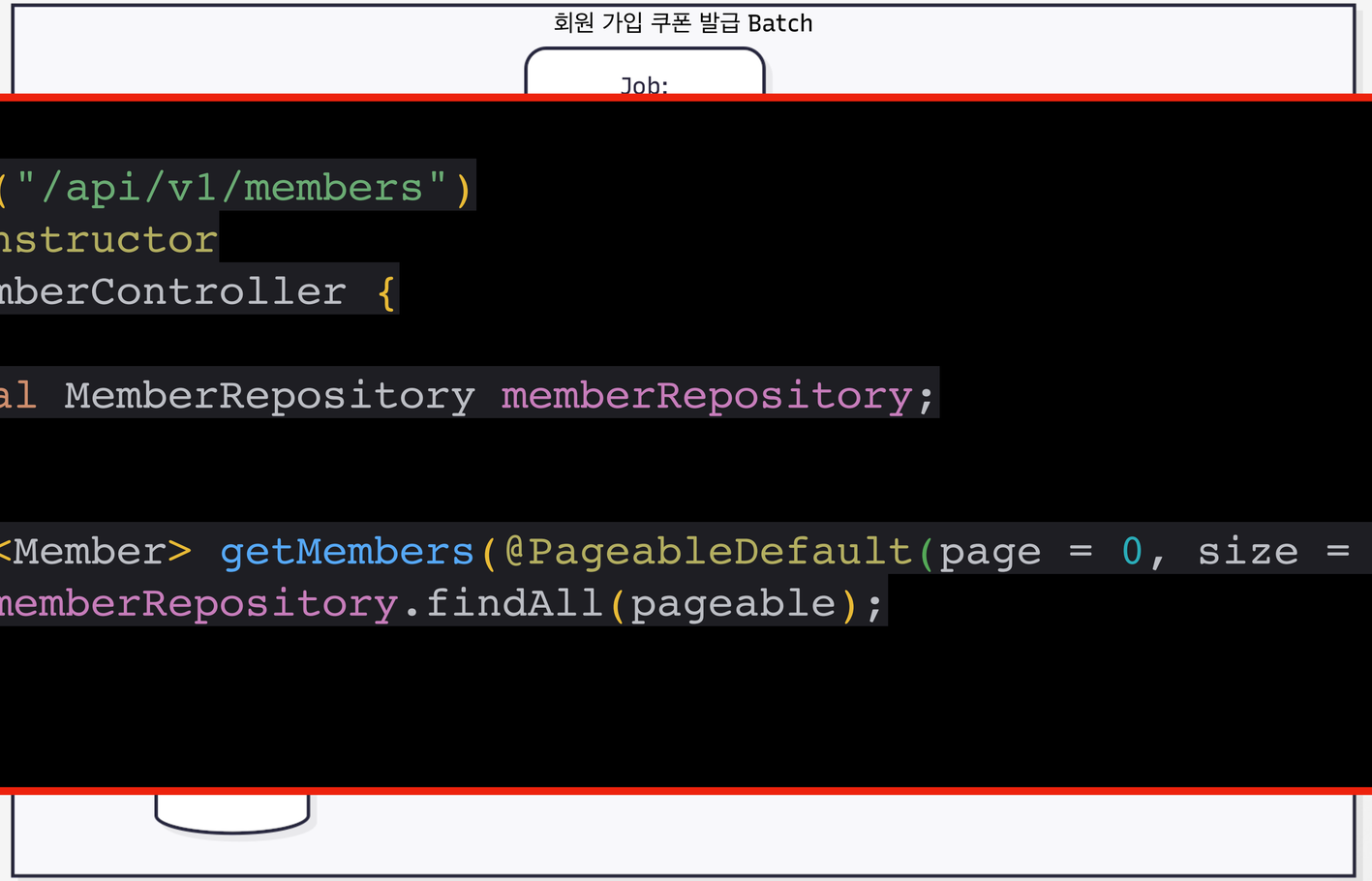
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;
}
```



회원 가입 쿠폰 발급 Job Flow




```java
@RestController
@RequestMapping("/api/v1/members")
@RequiredArgsConstructor
public class MemberController {

    private final MemberRepository memberRepository;

    @GetMapping
    public Page<Member> getMembers(@PageableDefault(page = 0, size = 10) Pageable pageable) {
        return memberRepository.findAll(pageable);
    }
}
```

회원 가입 쿠폰 발급 Job Flow

```
@RestController
@RequestMapping("/api/v
@RequiredArgsConstructo
public class MemberCont

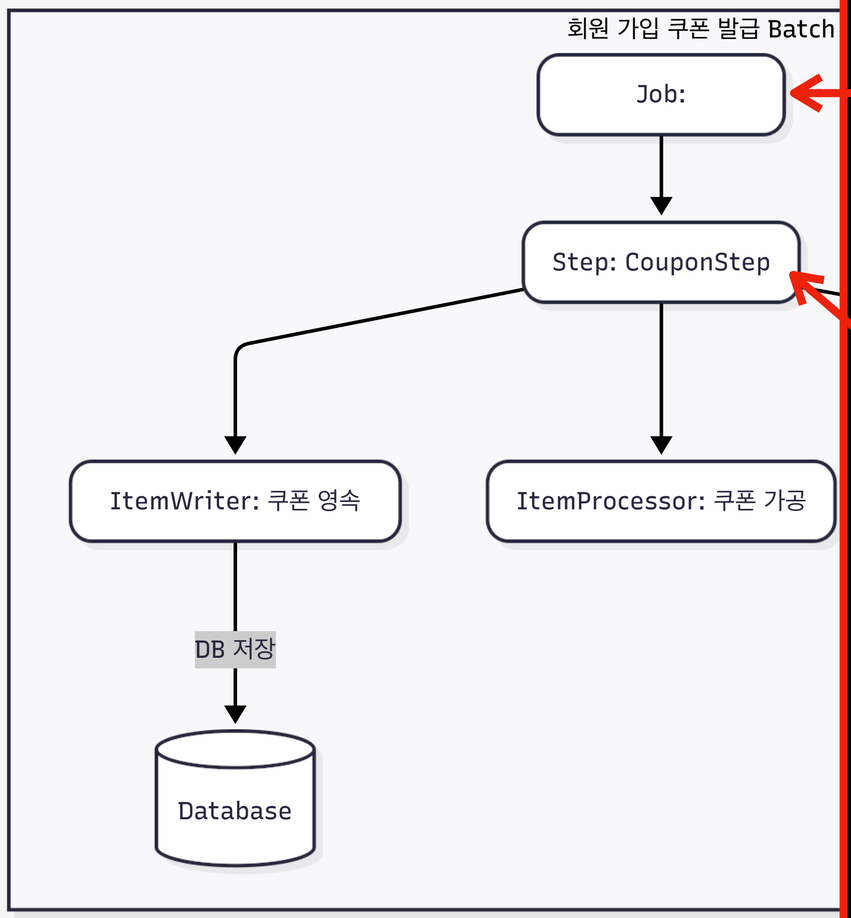
    private final Membe

    @GetMapping
    public Page<Member>
        return memberRe
    }
}
```

```
{
  "content": [
    {
      "id": 1,
      "name": "user1",
      "email": "user1@example.com"
    }
  ],
  "page": 10,
  "size": 10,
  "totalPages": 10,
  "totalElements": 100,
  "first": true,
  "last": false
}
```

```
0) Pageable pageable) {
```

회원 가입 쿠폰 발급 Job Flow

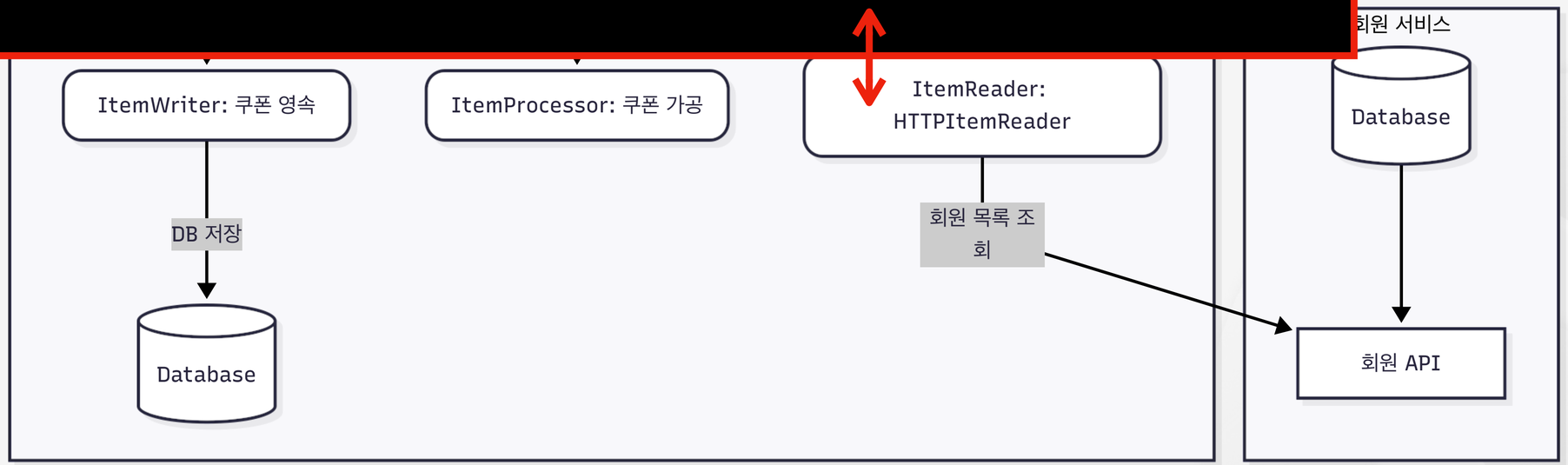



```java
@Bean
public Job couponJob(
        Step couponStep
) {
    return new JobBuilder("couponJob", jobRepository)
            .incrementer(new RunIdIncrementer())
            .start(couponStep)
            .build();
}

@Bean
public Step couponStep(
        ItemReader<MemberResponse> couponReader
) {
    return new StepBuilder("couponStep", jobRepository)
            .<MemberResponse, Coupon>chunk(chunkSize, transactionManager)
            .reader(couponReader)
            .processor(couponProcessor())
            .writer(couponWriter())
            .build();
}
```

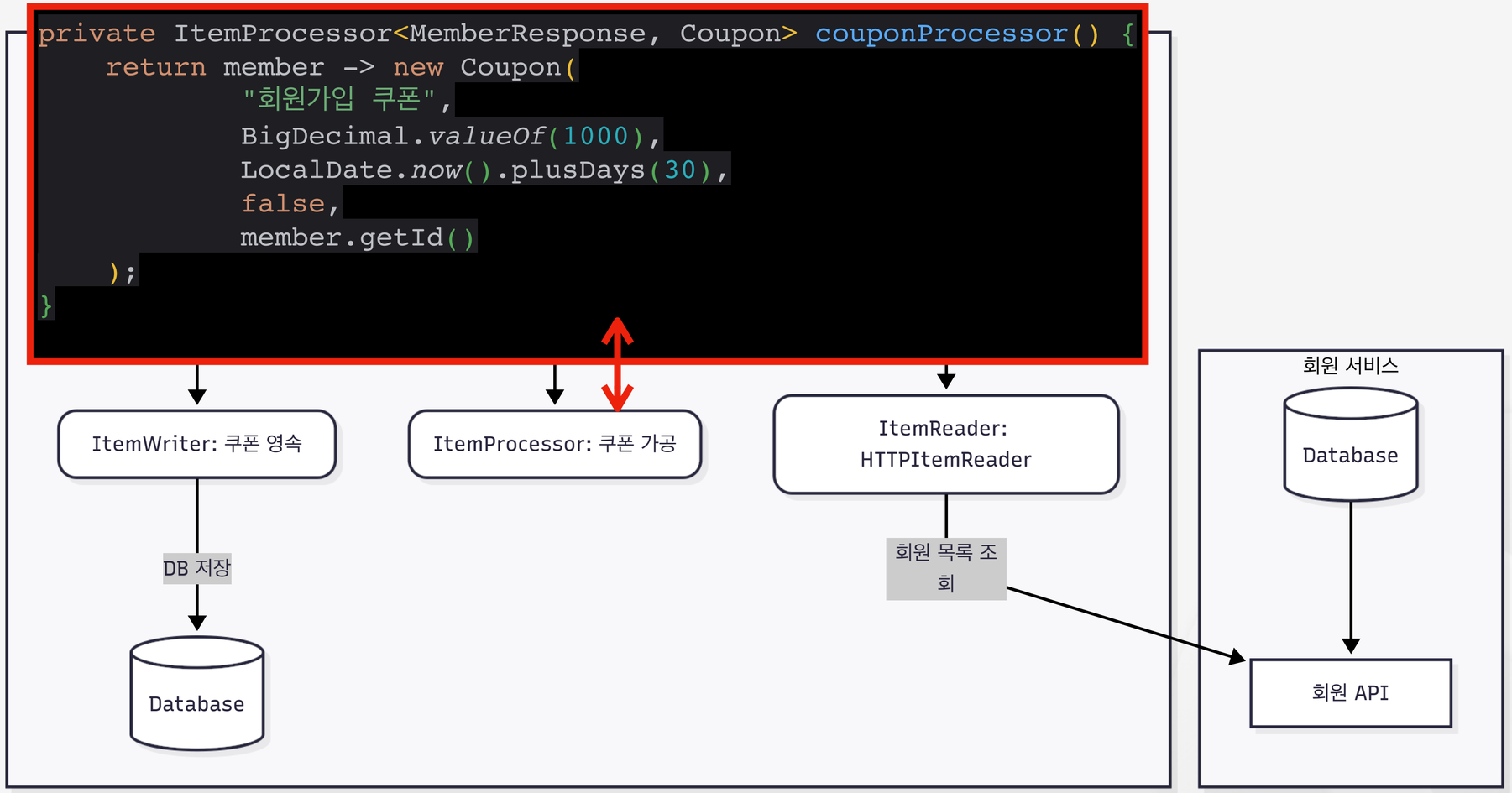
회원 가입 쿠폰 발급 Job Flow

```java
@Bean
public ItemReader<MemberResponse> couponReader(RestTemplate restTemplate) {
    return new HttpPageItemReaderBuilder<MemberResponse>()
            .baseUrl("http://localhost:8080/api/v1/members")
            .pageSize(chunkSize)
            .restTemplate(restTemplate)
            .responseType(new ParameterizedTypeReference<>() {})
            .build();
}
```



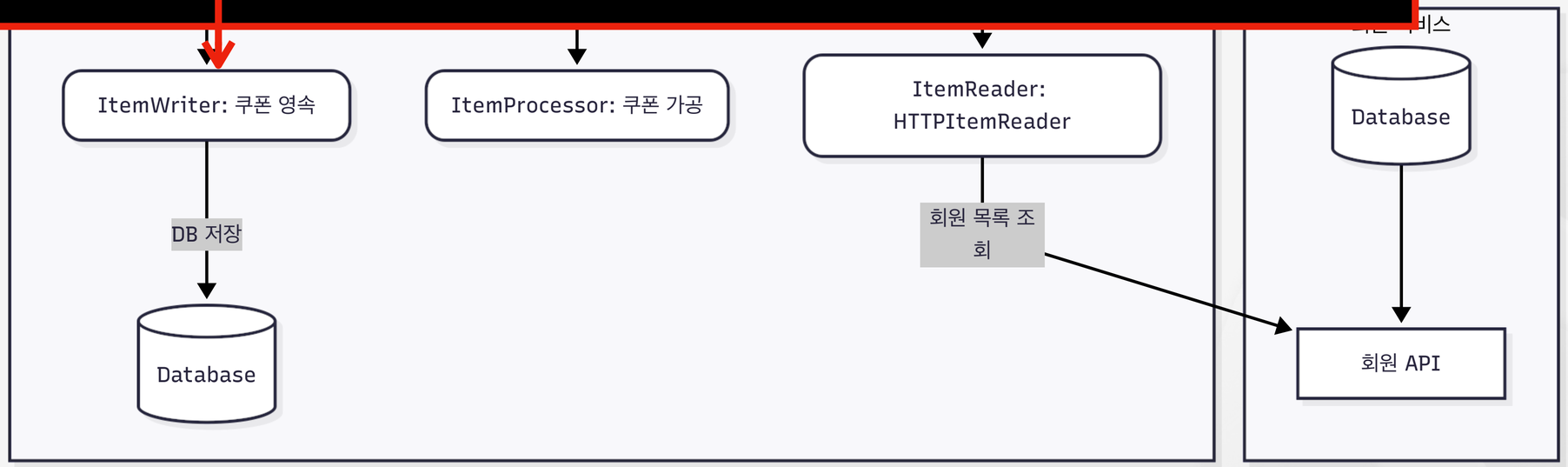
회원 가입 쿠폰 발급 Job Flow

```
private ItemProcessor<MemberResponse, Coupon> couponProcessor() {
    return member -> new Coupon(
            "회원가입 쿠폰",
            BigDecimal.valueOf(1000),
            LocalDate.now().plusDays(30),
            false,
            member.getId()
    );
}
```

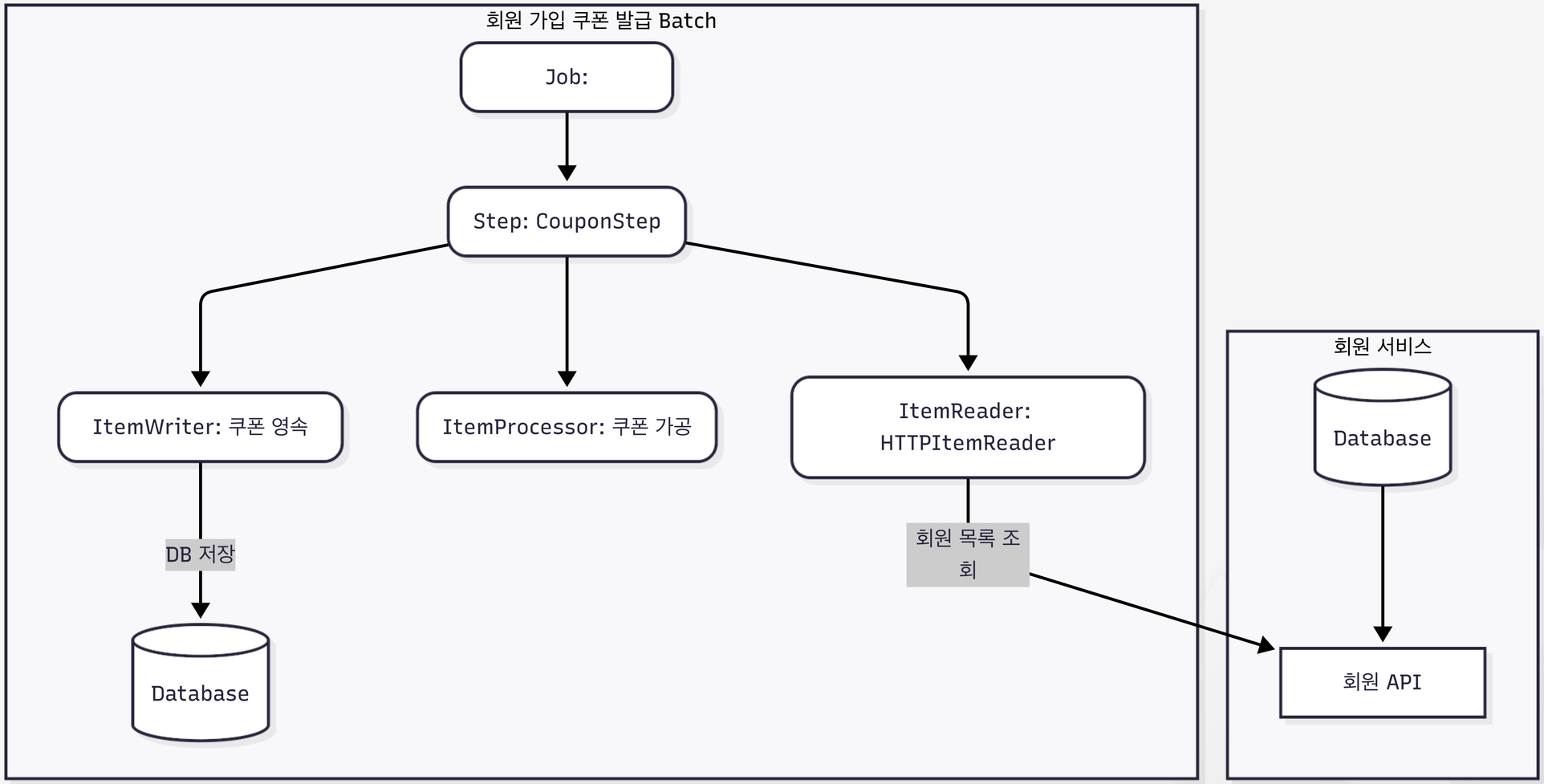


회원 가입 쿠폰 발급 Job Flow

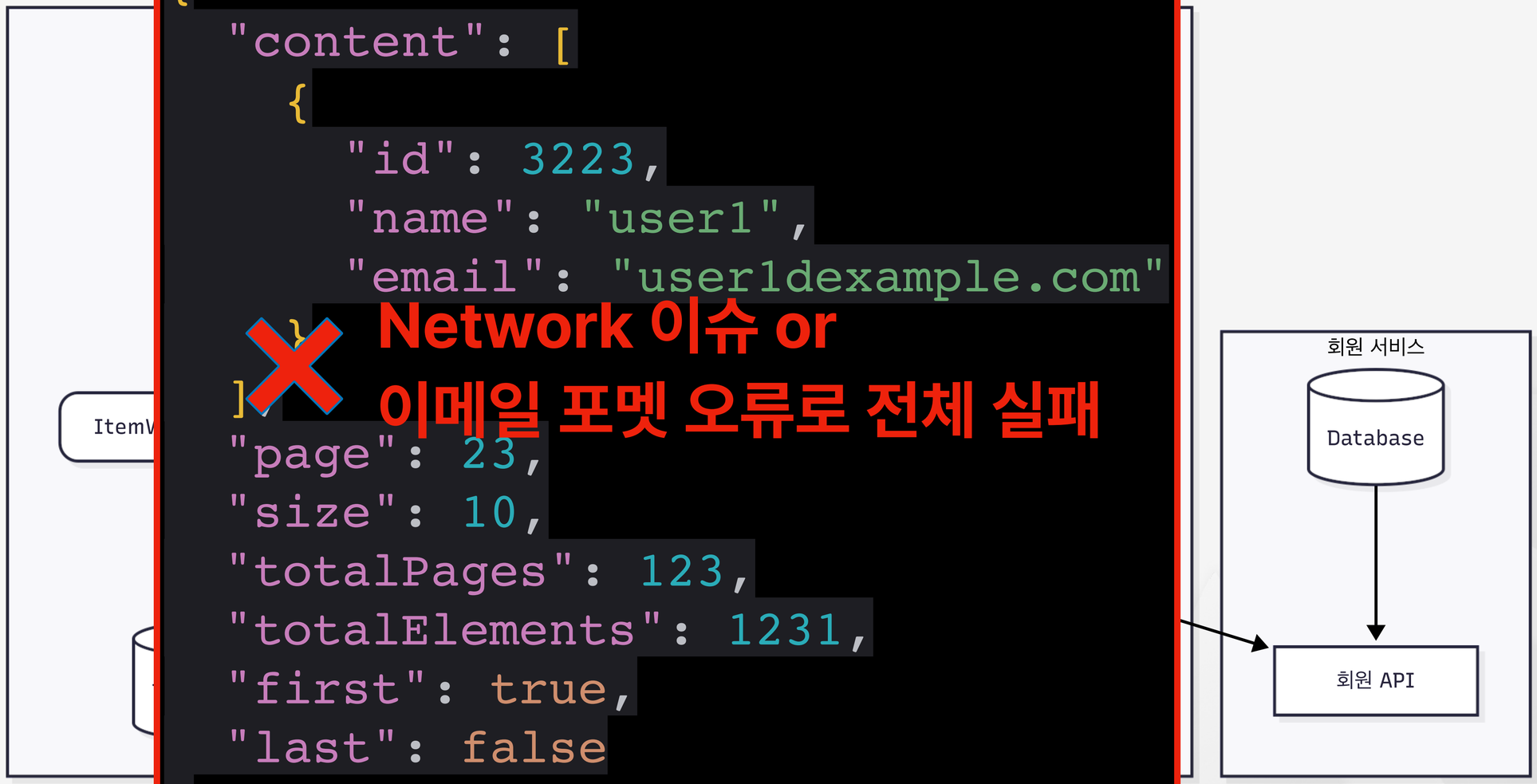
```java
@Bean
public JpaItemWriter<Coupon> couponWriter() {
    return new JpaItemWriterBuilder<Coupon>()
            .entityManagerFactory(entityManagerFactory)
            .build();
}
```



회원 가입 쿠폰 발급 Job 심화



회원 가입 쿠폰 발급 Job 실화



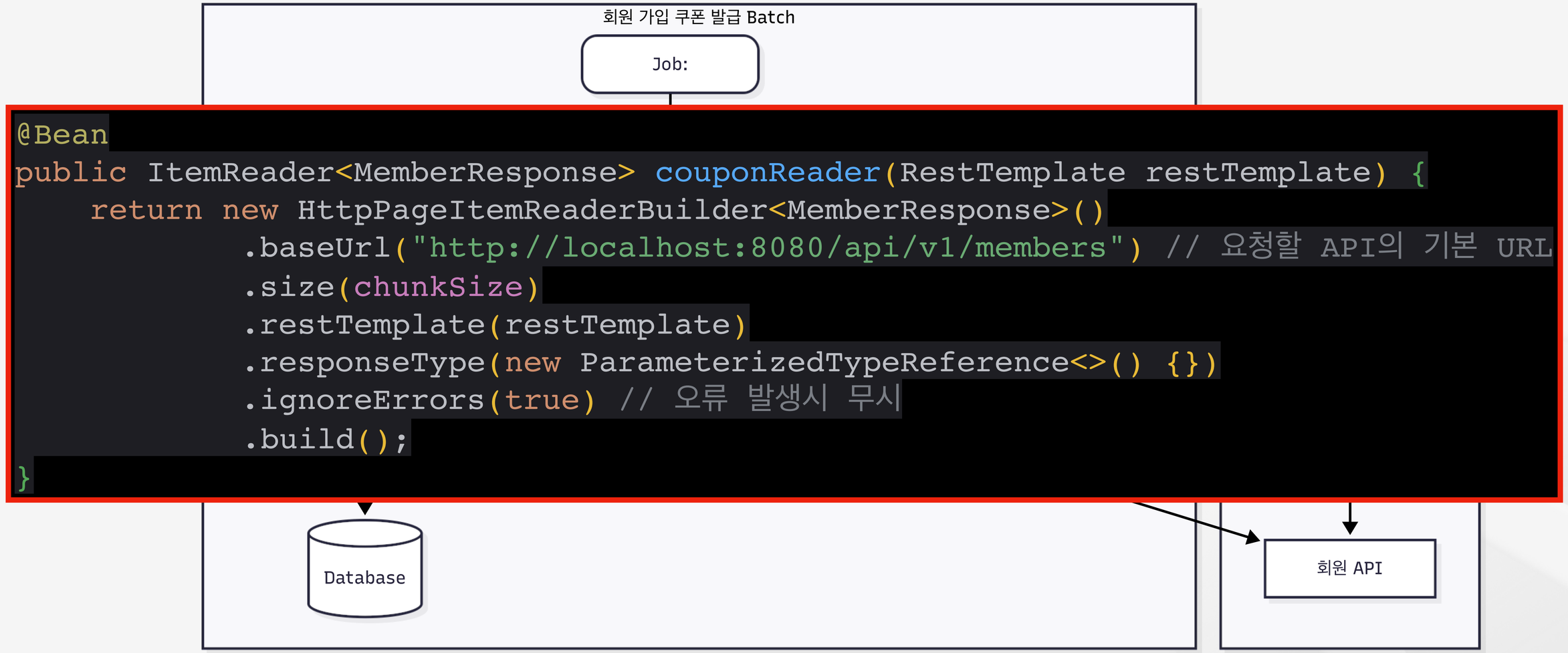

```
{
  "content": [
    {
      "id": 3223,
      "name": "user1",
      "email": "user1dexample.com"
    }
  ],
  "page": 23,
  "size": 10,
  "totalPages": 123,
  "totalElements": 1231,
  "first": true,
  "last": false
}
```


Network 이슈 or
이메일 포멧 오류로 전체 실패

회원 가입 쿠폰 발급 Job 실화



회원 가입 쿠폰 발급 Job 심화

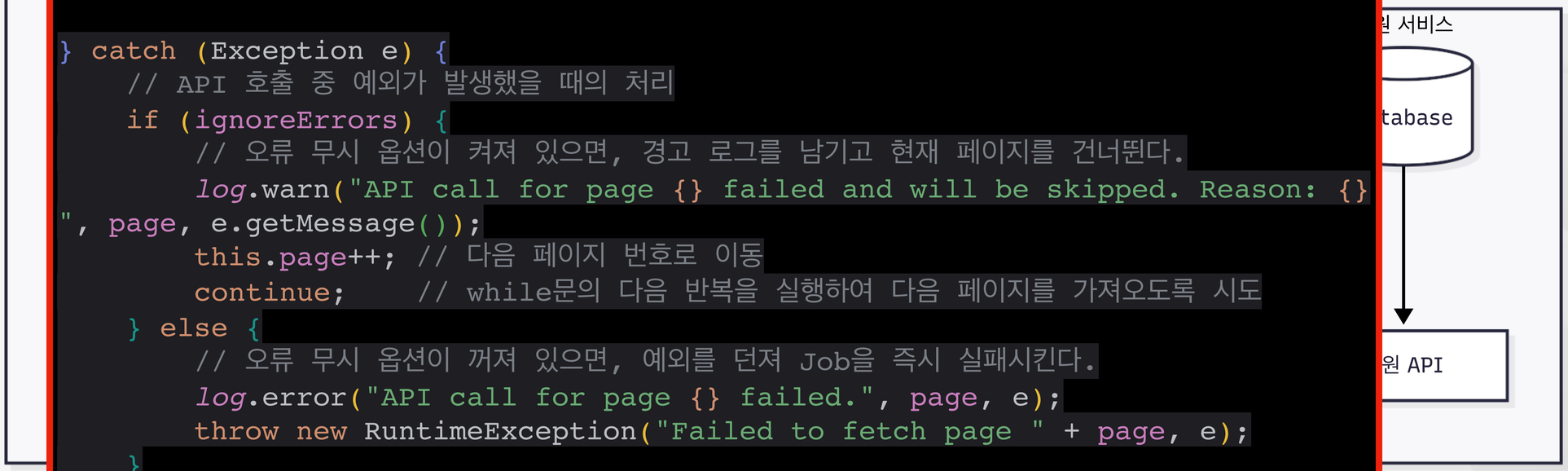



```java
@Bean
public ItemReader<MemberResponse> couponReader(RestTemplate restTemplate) {
    return new HttpPageItemReaderBuilder<MemberResponse>()
            .baseUrl("http://localhost:8080/api/v1/members") // 요청할 API의 기본 URL
            .size(chunkSize)
            .restTemplate(restTemplate)
            .responseType(new ParameterizedTypeReference<>() {})
            .ignoreErrors(true) // 오류 발생시 무시
            .build();
}
```

회원 가입 쿠폰 발급 Job 심화

```
PageResponse<T> pageResponse;
try {
    // 3. RestTemplate을 사용하여 API를 호출한다.
    ResponseEntity<PageResponse<T>> response = restTemplate.exchange(
            uriBuilder.toUriString(),
            HttpMethod.GET,
            null, // 요청 본문(body)은 없음
            responseType
    );
    pageResponse = response.getBody();

} catch (Exception e) {
    // API 호출 중 예외가 발생했을 때의 처리
    if (ignoreErrors) {
        // 오류 무시 옵션이 켜져 있으면, 경고 로그를 남기고 현재 페이지를 건너뛴다.
        log.warn("API call for page {} failed and will be skipped. Reason: {}
", page, e.getMessage());
        this.page++; // 다음 페이지 번호로 이동
        continue;    // while문의 다음 반복을 실행하여 다음 페이지를 가져오도록 시도
    } else {
        // 오류 무시 옵션이 꺼져 있으면, 예외를 던져 Job을 즉시 실패시킨다.
        log.error("API call for page {} failed.", page, e);
        throw new RuntimeException("Failed to fetch page " + page, e);
    }
}
```



외부 서버로 부터 데이터 읽어 오기 정리

- HTTP Page Item Reader의 가치
  - 시스템 간 결합도 완화: 외부 시스템의 데이터베이스에 직접 접근하는 대신, HTTP API 통신을 통해 데이터를 읽어옵니다. 이는 우리 배치 시스템과 외부 시스템 간의 의존성을 낮춥니다.
  - 안정성 및 보안 강화: API는 명확한 계약을 통해 필요한 데이터만 제공하므로, 의도치 않은 데이터 변경을 방지하고 보안을 강화합니다.데이터 무결성 및 보안 강화
- 실무적 고려사항: 재처리가 가능한 배치
  - 실패 시나리오 분석: 배치 작업은 다양한 이유로 실패할 수 있습니다. 특히 대량의 데이터를 다룰 때, 네트워크 이슈나 데이터 포맷 오류는 빈번하게 발생합니다.
  - 도메인 기반의 실패 전략: 모든 실패를 동일하게 처리할 필요는 없습니다.
    - 부분 실패 허용: 쿠폰 발급과 같이 데이터 정합성이 극도로 중요하지 않은 도메인에서는 일부 데이터의 오류를 무시하고 나머지 작업을 진행하는 부분 실패 전략이 효율적일 수 있습니다.
    - 전체 실패 필수: 가맹점 정산과 같이 금액과 관련된 민감한 도메인에서는 정확한 처리를 위해 전체 실패를 유도하고, 오류를 수정한 후 재처리하는 것이 바람직합니다.
  - 효율적인 재처리: 장시간 실행되는 배치 작업의 경우, 처음부터 다시 시작하는 대신 실패한 부분만 재처리할 수 있는 로직을 설계해야 합니다. 이는 배치 완료 시간을 단축하고 운영 효율성을 높이는 핵심 요소입니다.

Part 3 Spring Batch 실무에서의 배치

01

재처리가 가능한 Batch

- 멱등성 배치 어플리케이션 개발
- 데이터 누락이 케이스 대응

02

HTTP Page Item Reader

- 외부 데이터를 읽어 오는 방법
- 시스템의 강결합 해결

03

Batch Test 작성하는 노하우

- 다양한 테스트 케이스를 확보 하기 위한 방법
- 실무에서 의미 있는 테스트 코드를 작성하는 방법

개요 및 테스트의 중요성

- Spring Batch 테스트의 중요성 및 철학
  - 배치 애플리케이션의 핵심: 대규모 데이터 처리, 데이터 정합성 및 무결성 절대적 중요
  - 테스트의 목표: 기능 검증 넘어 데이터 일관성 보장, 예상치 못한 오류 사전 방지
  - Spring Batch 테스트 철학: 통합 테스트에 중점
- 주요 테스트 전략
  - 전체 Job 실행 (End-to-End 테스트): JobLauncherTestUtils.launchJob()
    - 전체 Job 실행 흐름을 처음부터 끝까지 검증
    - 데이터베이스, 파일 시스템 등 외부 시스템과의 통합 및 비즈니스 로직 수행 검증
    - 권장되는 가장 중요한 방법
- 단일 컴포넌트 단위 테스트 (Step 단위 테스트): JobLauncherTestUtils.launchStep()
  - 복잡한 Job의 개별 Step을 독립적으로 테스트
  - 특정 Step 오류 격리 및 디버깅 용이
  - 테스트의 "고립성(Isolation)" 원칙 준수
  - 전체 Job 실행의 비효율성 극복

Batch 테스트의 핵심 JobLauncherTestUtils

- **JobLauncherTestUtils**
  - 핵심 역할: Spring Batch 테스트 환경에서 Job과 Step을 실행하는 핵심 유틸리티 클래스.
  - 사용법:
    - Spring Test Framework의 일부.
    - 의존성 주입를 통해 ApplicationContext에서 주입받아 사용.
- **@SpringBatchTest 어노테이션**
  - 기능: 테스트 클래스에 Spring Batch 관련 테스트 환경을 자동 설정.
  - 효과:
    - JobLauncherTestUtils와 JobRepositoryTestUtils를 자동으로 Bean으로 등록하여 즉시 사용 가능.
    - @StepScope, @JobScope와 같은 스코프가 설정된 Bean 테스트를 위해 필요한 리스너들을 자동 등록.
    - Job Parameter나 Step Execution Context를 동적으로 주입받는 컴포넌트 테스트에 필수적.

JobLauncherTestUtils 주요 메서드

메서드	설명	반환 타입
launchJob()	가장 기본적인 End-to-End 테스트를 위해 사용합니다. 내부적으로 고유한 JobParameter를 생성하여 Job을 시작합니다.	JobExecution
launchJob(JobParameters jobParameters)	jobParameters)특정 비즈니스 시나리오(예: 특정 날짜, 파일명 등)에 따라 Job의 동작을 테스트할 때 사용됩니다.	JobExecution
setJob(Job job)	유틸리티가 사용할 Job 인스턴스를 설정합니다.	void
launchStep(String stepName)	복잡한 Job에서 특정 Step의 로직만 격리하여 테스트할 때 사용됩니다. 전체 Job 실행의 비효율성을 줄이고 빠른 디버깅을 가능하게 합니다.	JobExecution
launchStep(String stepName, JobParameters jobParameters)	@StepScope 컴포넌트가 JobParameter를 주입받는 경우, 해당 Step을 올바른 파라미터와 함께 테스트할 때 사용됩니다.	JobExecution

Spring Batch 테스트 코드

```
@SpringBatchTest
@SpringBootTest(properties = {"spring.batch.job.enabled=false"})
@TestPropertySource(properties = {"args.payment-date=2023-01-05"})
@ActiveProfiles("test")
class PaymentStatisticsJobConfigTest {

    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;

    @Autowired
    private Job paymentStatisticsJob;
    ...
}
```

Spring Batch 테스트 코드

```java
@BeforeEach
void setUp() {
    // JobLauncherTestUtils에 테스트할 Job을 명시적으로 설정합니다.
    this.jobLauncherTestUtils.setJob(paymentStatisticsJob);
}

@Test
void paymentStatisticsJob_test() throws Exception {
    // given
//    JobParameters jobParameters = new JobParametersBuilder()
//            .addString("requestDate", "2023-01-03")
//            .toJobParameters();

    // when
    JobExecution jobExecution = jobLauncherTestUtils.launchJob();

    // then
    assertEquals(ExitStatus.COMPLETED, jobExecution.getExitStatus());
}
```

paymentStatisticsJob Given

```
// given
paymentSourceRepository.saveAll(
        List.of(
                new PaymentSource("사업자1", "10002000", new BigDecimal(“100”), …),
                new PaymentSource("사업자1", "10002000", new BigDecimal(“300"), …),
                new PaymentSource("사업자2", "2002231", new BigDecimal(“1000”), …)
        )
);
```

paymentStatisticsJob Then

```
// then
final List<PaymentDailyStatistics> paymentDailyStatistics =
        paymentDailyStatisticsRepository.findByPaymentDate(properties.getPaymentDate());
then(paymentDailyStatistics).hasSize(2);
then(paymentDailyStatistics).allSatisfy(dailySum -> {
    switch (dailySum.getBusinessRegistrationNumber()) {
        case "10002000" -> {
            then(dailySum.getCorpName()).isEqualTo("사업자1");
            then(dailySum.getAmount()).isEqualByComparingTo(new BigDecimal("400"));
        }
        case "2002231" -> {
            then(dailySum.getCorpName()).isEqualTo("사업자2");
            then(dailySum.getAmount()).isEqualByComparingTo(new BigDecimal("1000"));
        }
        default -> fail("예상치 못한 사업자 번호입니다: " + dailySum.getBusinessRegistrationNumber());
    }
});
```

테스트 코드 개선점 - 반복적인 코드

```
@SpringBatchTest
@SpringBootTest(properties = {"spring.batch.job.enabled=false"})
@TestPropertySource(properties = {"args.payment-date=2025-01-05"})
@ActiveProfiles("test")
class PaymentStatisticsJobConfigTest {

    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;

    @Autowired
    private Job paymentStatisticsJob;

    @Autowired
    private ArgumentProperties properties;

    @Autowired
    private PaymentDailyStatisticsRepository paymentDailyStatisticsRepository;

    @Autowired
    private PaymentSourceRepository paymentSourceRepository;
}
```

테스트 코드 개선점 - 반복적인 의존성

```
// given
paymentSourceRepository.saveAll(
        List.of(
                new PaymentSource("사업자1", "10002000", new BigDecimal(“100”), …),
                new PaymentSource("사업자1", "10002000", new BigDecimal(“300"), …),
                new PaymentSource("사업자2", "2002231", new BigDecimal(“1000”), …)
        )
);
```

테스트 코드 개선점 - 테스트 코드만을 위한 코드

```
// then
final List<PaymentDailyStatistics> paymentDailyStatistics =
        paymentDailyStatisticsRepository.findByPaymentDate(properties.getPaymentDate());
then(paymentDailyStatistics).hasSize(2);
then(paymentDailyStatistics).allSatisfy(dailySum -> {
    switch (dailySum.getBusinessRegistrationNumber()) {
        case "10002000" -> {
            then(dailySum.getCorpName()).isEqualTo("사업자1");
            then(dailySum.getAmount()).isEqualByComparingTo(new BigDecimal("400"));
        }
        case "2002231" -> {
            then(dailySum.getCorpName()).isEqualTo("사업자2");
            then(dailySum.getAmount()).isEqualByComparingTo(new BigDecimal("1000"));
        }
        default -> fail("예상치 못한 사업자 번호입니다: " + dailySum.getBusinessRegistrationNumber());
    }
});
```

메서드는 실제 사용 X

BatchTestSupport을 통한 반복적인 의존성 개선

```
@SpringBatchTest
@SpringBootTest(properties = {"spring.batch.job.enabled=false"})
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@ActiveProfiles("test")
public abstract class SpringBatchTestSupport {
    @Autowired
    protected JobLauncherTestUtils jobLauncherTestUtils;
    @Autowired
    protected EntityManagerFactory entityManagerFactory;
    @Autowired
    protected Job paymentStatisticsJob;
    @Autowired
    protected ArgumentProperties properties;
    …
}
```

BatchTestSupport을 통한 유틸 기능 제공

```java
public abstract class SpringBatchTestSupport {
    protected JobExecution jobExecution;
    protected EntityManager entityManager;
    protected JPAQueryFactory query;

    @BeforeEach
    void setUp() {
        this.entityManager = entityManagerFactory.createEntityManager();
        this.query = new JPAQueryFactory(this.entityManager);
    }

    protected void launchJob(Job job, JobParameters jobParameters) throws Exception {
        this.jobLauncherTestUtils.setJob(job);
        this.jobExecution = this.jobLauncherTestUtils.launchJob(jobParameters);
    }

    protected void launchJob(Job job) throws Exception {
        launchJob(job, this.jobLauncherTestUtils.getUniqueJobParameters());
    }

    protected void thenBatchCompleted() {
        assertThat(jobExecution).isNotNull();
        assertThat(jobExecution.getStatus()).isEqualTo(BatchStatus.COMPLETED);
    }

    protected <T> List<T> saveAll(List<T> entities) {
        EntityManager entityManager = entityManagerFactory.createEntityManager();
        …
        return entities;
    }
}
```

BatchTestSupport을 통한 코드 간결화

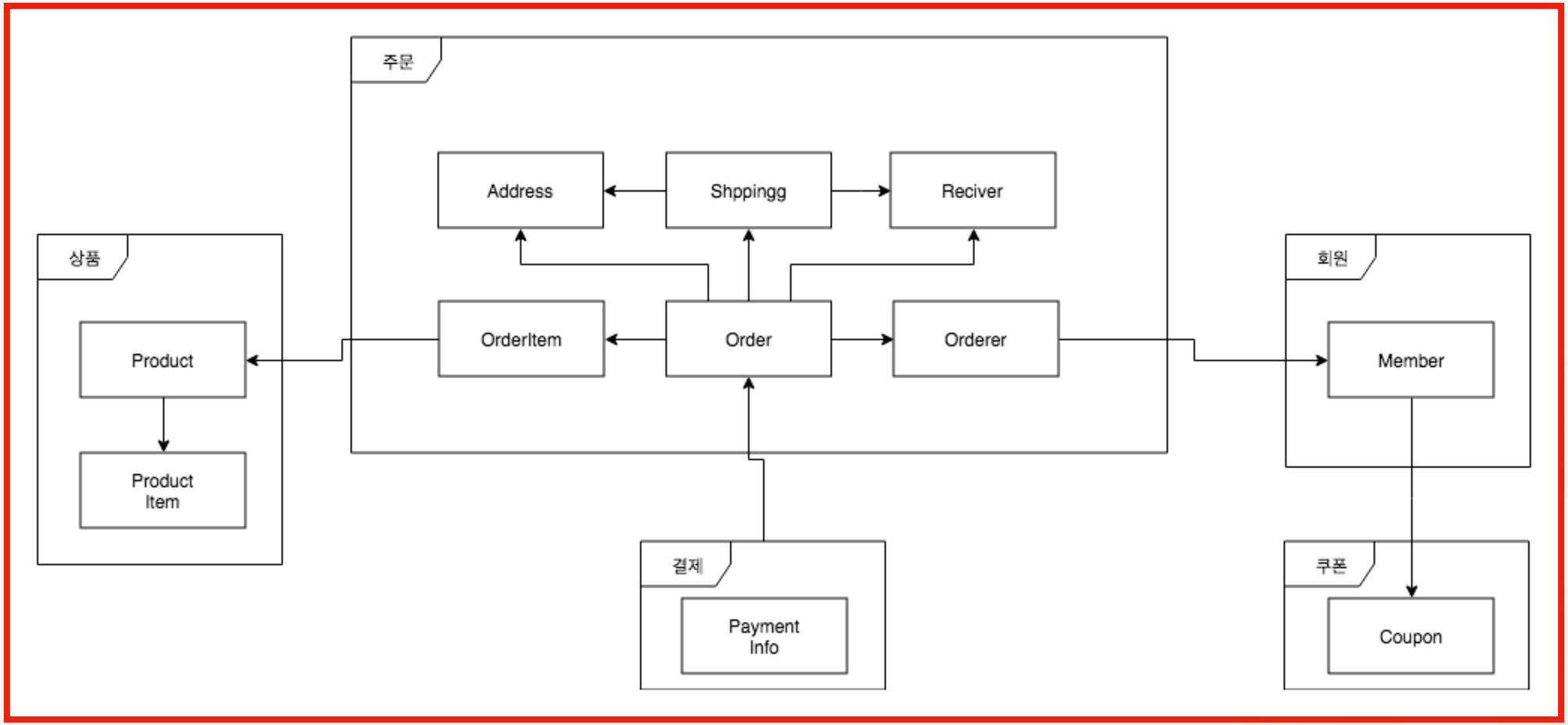
```
@TestPropertySource(properties = {"args.payment-date=2025-01-05"})
class PaymentStatisticsJobConfigTest extends SpringBatchTestSupport {

    @Test
    void paymentStatisticsJob_test() throws Exception {
        // given
        saveAll(
                List.of(
                        new PaymentSource("사업자1", "10002000", new BigDecimal("100"), ...)),
                        new PaymentSource("사업자1", "10002000", new BigDecimal("300"), ...)),
                        new PaymentSource("사업자2", "2002231", new BigDecimal("1000"), ...))
                )
        );

        // when
        launchJob(paymentStatisticsJob);

        // then
        thenBatchCompleted();
        final List<PaymentDailyStatistics> dailyStatistics = query.selectFrom(paymentDailyStatistics)
                .where(paymentDailyStatistics.paymentDate.eq(properties.getPaymentDate()))
                .fetch();
        ...
    }
}
```

복잡하고 다양한 데이터 셋업의 어려움



복잡하고 다양한 데이터 셋업의 어려움

```
// Given: 테스트에 필요한 복잡한 데이터 셋업

// 1. 회원 정보 생성
        Member member = new Member(
                "회원1",
                "user1@test.com",
    ... // 추가적인 회원 정보 필드
);
        memberRepository.save(member);

// 2. 상품 정보 생성
        Product product = new Product(
                "상품A",
                new BigDecimal("10000"),
                100, // 재고
    ... // 추가적인 상품 정보 필드
);
        productRepository.save(product);

        ProductItem productItem = new ProductItem(
                product,
    ... // 상품 아이템 관련 필드
);
        productItemRepository.save(productItem);

// 3. 쿠폰 정보 생성
        Coupon coupon = new Coupon(
                "5000원 할인",
                new BigDecimal("5000"),
    ... // 추가적인 쿠폰 정보 필드
);
        couponRepository.save(coupon);
```

```
// 4. 결제 정보 생성
PaymentInfo paymentInfo = new PaymentInfo(
        new BigDecimal("5000"), // 최종 결제 금액
... // 결제 관련 필드
);
paymentInfoRepository.save(paymentInfo);

// 5. 주문, 주소, 배송, 주문 상품 등 복합적인 정보 생성
Address address = new Address(
        "서울시 강남구",
        "12345",
...
);
Shipping shipping = new Shipping(
        "빠른 배송",
        "010-1234-5678",
...
);
Orderer orderer = new Orderer(
        member,
...
);

Order order = new Order(member,… paymentInfo,
... // 주문 관련 필드
);
orderRepository.save(order);

// 6. 주문 상품 정보 생성
OrderItem orderItem = new OrderItem(
        order,
        product,
        1, // 수량
...
);
orderItemRepository.save(orderItem);
```

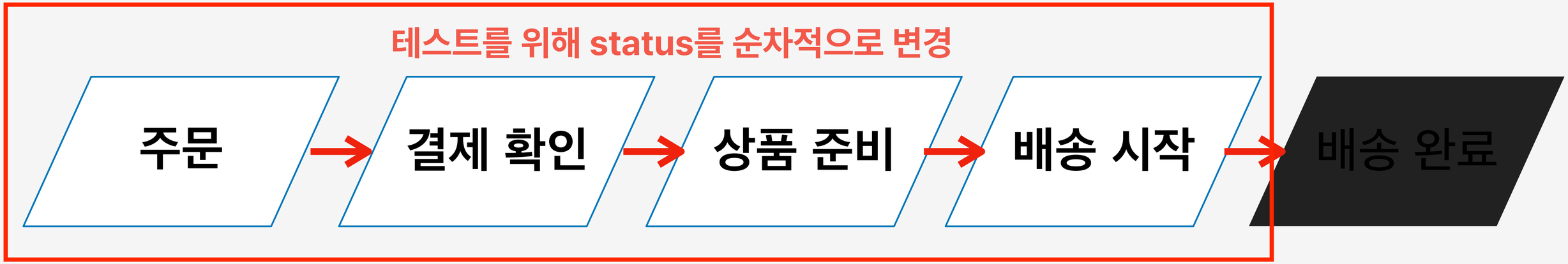
특정 시점의 데이터 셋업의 어려움



특정 시점의 데이터 셋업의 어려움



특정 시점의 데이터 셋업의 어려움

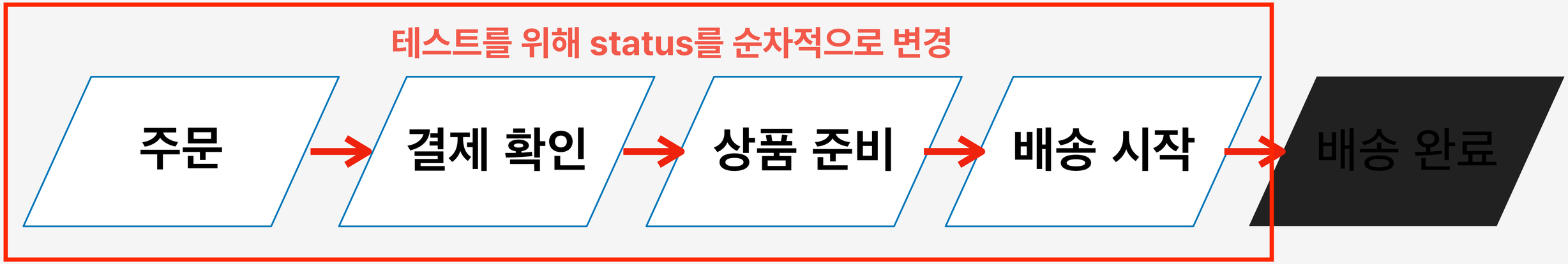



```java
@Test
void productPreparationToDeliveryStartStatusChangeTest() {
    // given
    String orderNumber = "order-number-123";

    Order order = new Order(orderNumber); // 주문 생성
    order.updateStatusCompletePayment(); // 결제 완료 상태 변경
    order.updateStatusProductPreparation(); // 상품 준비 상태 변경
    // 데이터 셋업 완료

    // when
    order.updateStatusDeliveryStarted(); // 배송 시작 상태 변경
}
```

특정 시점의 데이터 셋업의 어려움

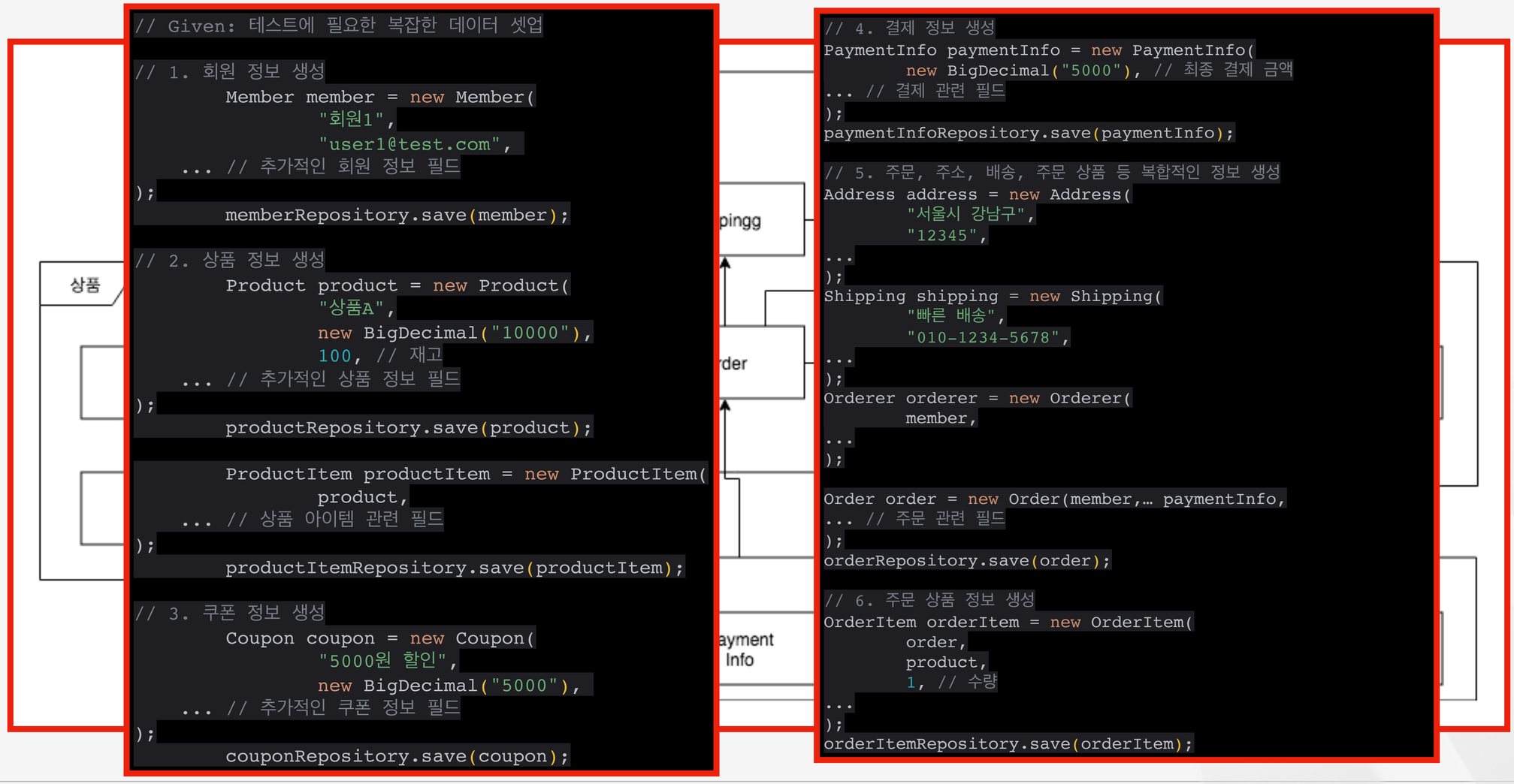



```
@Test
void productPreparationToDeliveryStartStatusChangeTest() {
    // given
    String orderNumber = "order-number-123";

    Order order = new Order(orderNumber); // 주문 상태 생성
    order.updateStatusCompletePayment(); // 결제 완료 상태 변경
    order.updateStatusProductPreparation(); // 상품 준비 상태 변경
    // 데이터 셋업 완료

    // when
    order.updateStatusDeliveryStarted(); // 배송 시작 상태 변경
}
```

특정 시점의 데이터 셋업의 어려움 극복 방법




```
// Given: 테스트에 필요한 복잡한 데이터 셋업

// 1. 회원 정보 생성
        Member member = new Member(
                "회원1",
                "user1@test.com",
    ... // 추가적인 회원 정보 필드
);
        memberRepository.save(member);

// 2. 상품 정보 생성
        Product product = new Product(
                "상품A",
                new BigDecimal("10000"),
                100, // 재고
    ... // 추가적인 상품 정보 필드
);
        productRepository.save(product);

        ProductItem productItem = new ProductItem(
                product,
    ... // 상품 아이템 관련 필드
);
        productItemRepository.save(productItem);

// 3. 쿠폰 정보 생성
        Coupon coupon = new Coupon(
                "5000원 할인",
                new BigDecimal("5000"),
    ... // 추가적인 쿠폰 정보 필드
);
        couponRepository.save(coupon);
```

```
// 4. 결제 정보 생성
PaymentInfo paymentInfo = new PaymentInfo(
        new BigDecimal("5000"), // 최종 결제 금액
... // 결제 관련 필드
);
paymentInfoRepository.save(paymentInfo);

// 5. 주문, 주소, 배송, 주문 상품 등 복합적인 정보 생성
Address address = new Address(
        "서울시 강남구",
        "12345",
...
);
Shipping shipping = new Shipping(
        "빠른 배송",
        "010-1234-5678",
...
);
Orderer orderer = new Orderer(
        member,
...
);

Order order = new Order(member,… paymentInfo,
... // 주문 관련 필드
);
orderRepository.save(order);

// 6. 주문 상품 정보 생성
OrderItem orderItem = new OrderItem(
        order,
        product,
        1, // 수량
...
);
orderItemRepository.save(orderItem);
```

특정 시점의 데이터 셋업의 어려움 극복 방법

```sql
-- 회원 데이터 삽입
INSERT INTO member (member_id, name, email, created_at) VALUES (1, '회원1', 'user1@test.com', NOW());

-- 상품 데이터 삽입
INSERT INTO product (product_id, name, price, stock, created_at) VALUES (1, '상품A', 10000, 100, NOW());

-- 쿠폰 데이터 삽입
INSERT INTO coupon (coupon_id, name, discount_amount, created_at) VALUES (1, '5000원 할인', 5000, NOW());

-- 결제 정보 삽입
INSERT INTO payment_info (payment_info_id, amount, created_at) VALUES (1, 5000, NOW());

-- 주문 관련 데이터 삽입
INSERT INTO orderer (orderer_id, member_id, created_at) VALUES (1, 1, NOW());
INSERT INTO address (address_id, full_address, zipcode, created_at) VALUES (1, '서울시 강남구', '12345', NOW());
INSERT INTO shipping (shipping_id, orderer_id, address_id, status, created_at) VALUES (1, 1, 1, '상품 준비중', NOW());

-- 주문 데이터 삽입 (Member, Orderer, Shipping, Coupon, PaymentInfo와의 외래 키 관계 설정)
INSERT INTO `order` (order_id, member_id, orderer_id, shipping_id, coupon_id, payment_info_id, total_amount, status, created_at)
VALUES (1, 1, 1, 1, 1, 1, 5000, '결제 완료', NOW());

-- 주문 상품 데이터 삽입
INSERT INTO order_item (order_item_id, order_id, product_id, quantity, created_at) VALUES (1, 1, 1, 1, NOW());
```

특정 시점의 데이터 셋업의 어려움 극복 방법

```
DELETE FROM order_item;

DELETE FROM `order`;

DELETE FROM shipping;

DELETE FROM address;

DELETE FROM orderer;

DELETE FROM payment_info;

DELETE FROM coupon;

DELETE FROM product;

DELETE FROM member;
```

특정 시점의 데이터 셋업의 어려움 극복 방법

```
@Test
@SqlGroup({
        @Sql(value = "/sql/order-setup.sql", executionPhase =
Sql.ExecutionPhase.BEFORE_TEST_METHOD),
        @Sql(value = "/sql/order-cleanup.sql", executionPhase =
Sql.ExecutionPhase.AFTER_TEST_METHOD)
})
void orderProcessingJobTest() {
    // given
    // 데이터 셋업은 SQL 파일이 자동으로 처리

    // when
    // launchJob(orderProcessingJob);

    // then
    // 결과 검증
}
```

@Sql 어노테이션 활용

- @Sql 어노테이션 활용
  - Spring에서는 @Sql 어노테이션을 제공하여 이 과정을 자동화하고 테스트 코드의 **Given 절을 극적으로 간소화**합니다.
  - 테스트 전/후 자동 실행: @SqlGroup 안에 @Sql 어노테이션을 사용하여 테스트 **실행 전(BEFORE_TEST_METHOD)**에 데이터 셋업 스크립트를 실행하고, **테스트 실행 후(AFTER_TEST_METHOD)**에 데이터 정리 스크립트를 실행하도록 설정할 수 있습니다.
  - order-setup.sql: 이 파일은 테스트를 위해 필요한 모든 INSERT 문을 담고 있어, 복잡한 객체 생성 코드를 완벽하게 대체합니다.
  - order-cleanup.sql: 이 파일은 테스트가 끝난 후 데이터베이스를 깨끗하게 정리하는 DELETE 또는 TRUNCATE 문을 담고 있어, 다른 테스트에 영향을 주지 않는 **테스트의 격리성(Isolation)**을 보장합니다.
- @Sql 어노테이션 상세 활용법
  - @Sql 어노테이션은 단순히 SQL 파일을 실행하는 것을 넘어, **테스트 생명주기(lifecycle)**에 맞춰 다양한 방식으로 활용할 수 있습니다.
  - @Sql 속성
    - value: 실행할 SQL 파일의 경로를 지정합니다. 클래스패스 기준으로 상대 경로를 사용합니다.
    - executionPhase: SQL 스크립트가 실행될 시점을 정의합니다.
      - **BEFORE_TEST_METHOD** (기본값): @Test 메서드 실행 전에 스크립트를 실행합니다. 주로 테스트 데이터 셋업에 사용됩니다.
      - **AFTER_TEST_METHOD**: @Test 메서드 실행 후에 스크립트를 실행합니다. 주로 데이터 클린업에 사용됩니다.

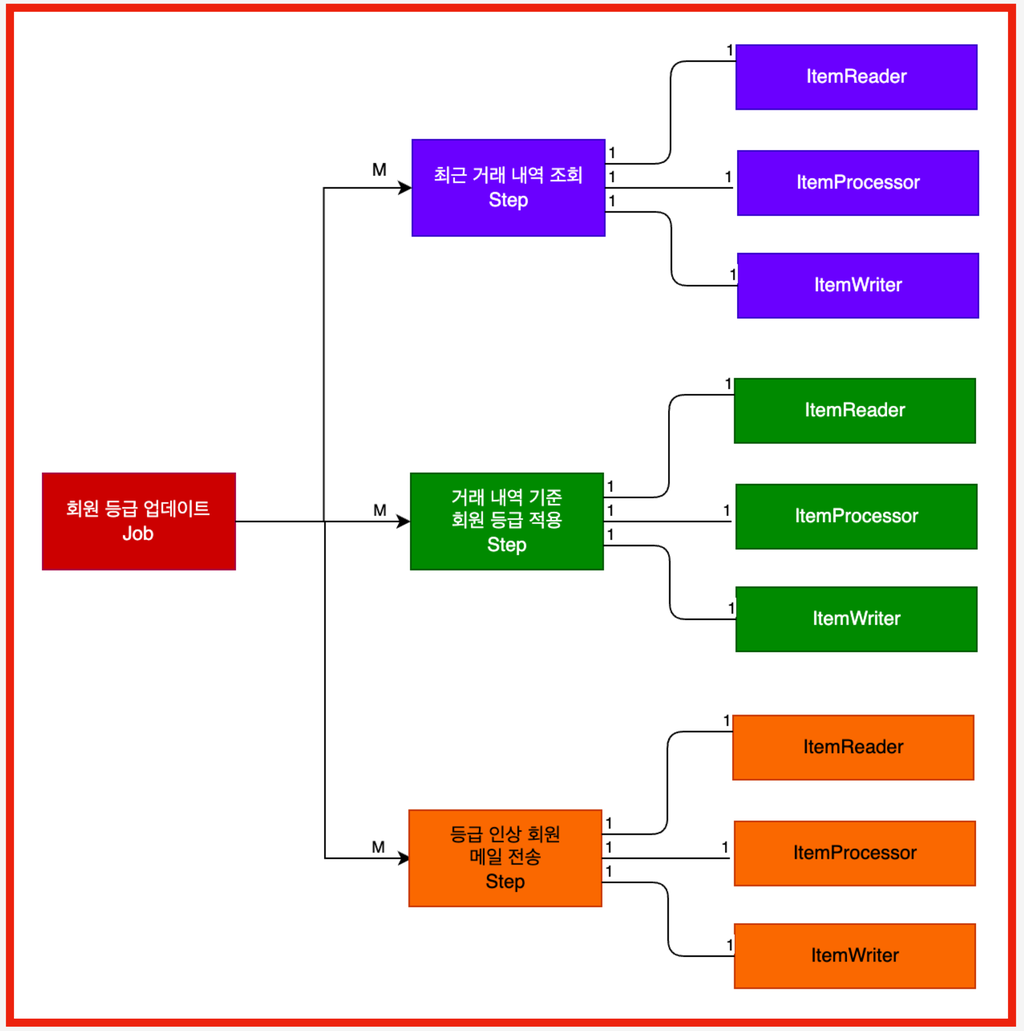
@Sql 어노테이션 활용

- @Sql의 동작 방식
  - @Sql 어노테이션은 Spring Test의 트랜잭션 관리와 함께 동작합니다. 기본적으로 @Test 메서드에는 트랜잭션이 적용되며, 테스트 실행 후 롤백됩니다. 하지만 **@Sql은 기본적으로 이 트랜잭션과는 별개로 동작하여, 테스트 전후에 실제 데이터베이스에 영구적인 변경을 가합니다.** 이를 통해 테스트 간 데이터 의존성을 완전히 차단할 수 있습니다.
- @SqlGroup을 통한 여러 스크립트 실행
  - 하나의 테스트 메서드에 **여러 개의 @Sql 어노테이션을 적용해야 할 경우, @SqlGroup을 사용합니다.** 이는 여러 개의 @Sql 어노테이션을 그룹화하는 컨테이너 역할을 합니다.
    - @SqlGroup({
    - @Sql(value = "/sql/payment-source-setup.sql", executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD),
    - @Sql(value = "/sql/payment-source-cleanup.sql", executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD)
    - })
  - 이 코드는 테스트 전에 setup.sql을 실행하고, 테스트가 끝난 뒤 cleanup.sql을 실행하도록 명시합니다. 이처럼 SQL 파일을 활용한 데이터 셋업은 테스트 코드의 가독성을 높이고, 개발 효율성을 극대화하며, 안정적인 테스트 환경을 구축하는 매우 효과적인 방법입니다.

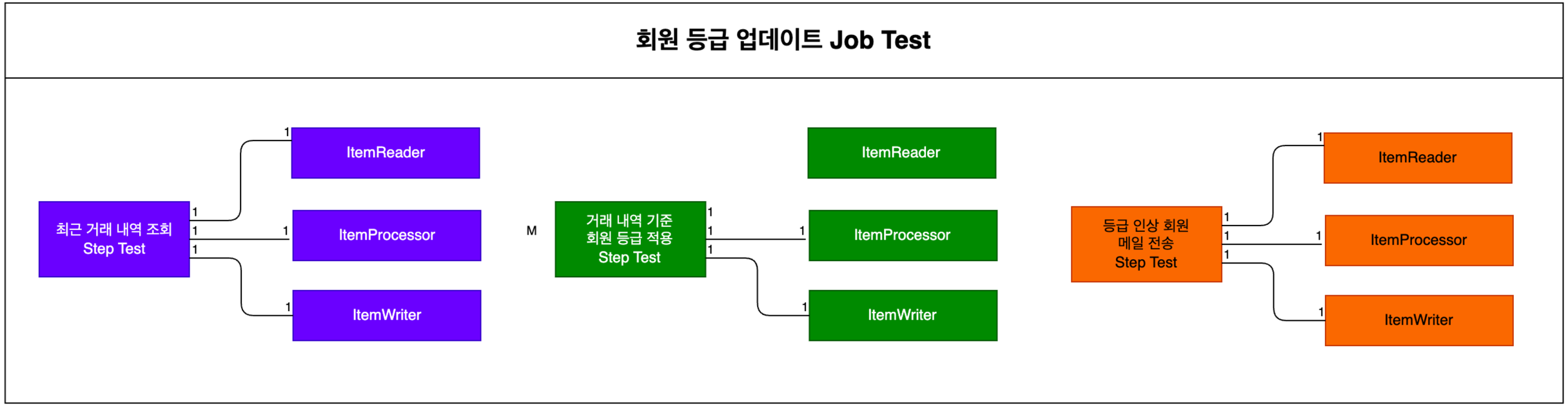
SQL 기반 데이터 셋업의 단점

- 변경에 취약하고 디버깅이 어렵다
  - 컬럼명/엔티티 변경: JPA 엔티티를 기반으로 테스트 코드를 작성하면, 컬럼명이나 엔티티 구조가 변경될 때 컴파일 타임에 오류를 즉시 감지할 수 있습니다. 하지만 SQL 파일은 단순 문자열 기반이므로, DB 스키마가 변경되어도 런타임에만 오류가 발생합니다. 이는 개발자가 변경사항을 즉시 알아차리기 어렵게 만들어 유지보수 비용을 높입니다.
- 독성이 낮고 의미 전달이 어렵다
  - 의미 전달의 한계: 객체 기반 코드에서는 변수명, 메서드명을 통해 데이터의 의미와 의도를 명확하게 전달할 수 있습니다. 예를 들어 lockedMember라는 변수명은 이 객체가 '락이 걸린 회원'임을 즉시 알 수 있게 해줍니다. 하지만 SQL 파일은 단순 INSERT 문이므로, 데이터의 맥락을 파악하기 어렵습니다. 예를 들어, 특정 상태를 표현하기 위해 숫자를 사용한 경우, 그 숫자가 무엇을 의미하는지 주석이나 문서를 따로 확인해야 합니다.
- 복잡한 비즈니스 로직 테스트의 한계
  - 동적인 데이터 생성의 어려움: SQL 파일은 정적인 데이터 셋업에 최적화되어 있습니다. 특정 비즈니스 로직에 따라 동적으로 변경되는 데이터를 테스트해야 할 경우, 예를 들어 현재 시각을 기준으로 만료되는 쿠폰 데이터를 생성할 때 SQL 파일만으로는 한계가 있습니다.
  - 특정 시나리오 재현의 제약: 여러 객체의 복잡한 상호작용을 테스트해야 할 때, 예를 들어 주문 상태를 '배송 중'으로 변경하는 시나리오를 재현하려면 SQL 파일만으로는 모든 중간 상태를 완벽하게 표현하기 어렵습니다. 객체 메서드를 이용하는 방식이 더 직관적일 수 있습니다.

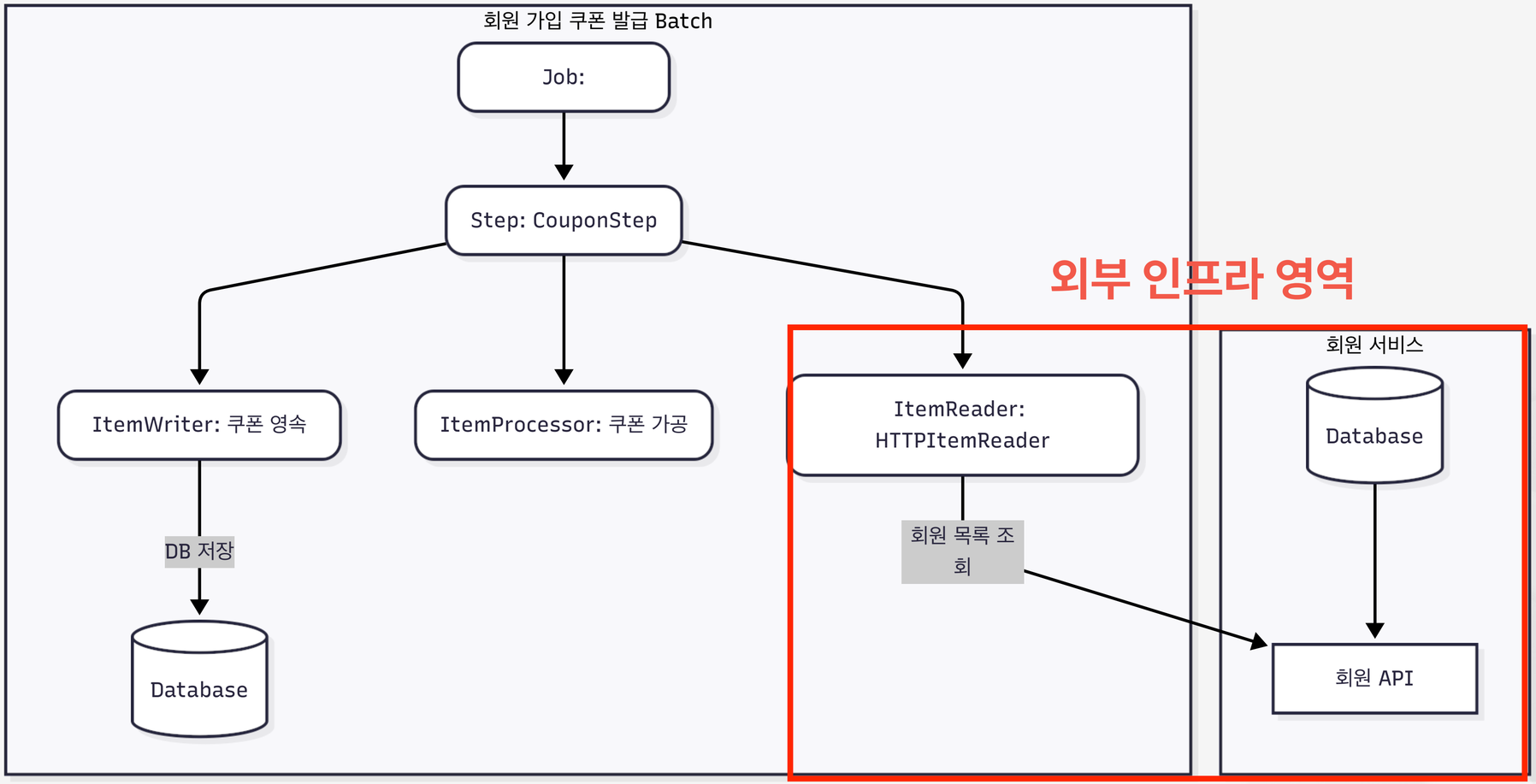
복잡한 다양한 Job 테스트의 어려움 문제



복잡한 다양한 Job 테스트의 어려움 문제



외부 인프라를 의존하는 테스트의 어려움 문제



MockServer 기반 테스트 방법

- Mock Server란?
  - 외부 API 서버의 역할을 대신하여 가짜(Mock) 응답을 보내주는 서버.
  - 실제 외부 API에 의존하지 않고 테스트 환경을 구축하는 핵심 도구.
- Mock Server의 장점
  - 외부 시스템 의존성 제거: 네트워크 상태, 외부 API의 장애 등 외부 요인으로부터 테스트를 격리하여 안정적이고 예측 가능한 테스트를 가능하게 합니다.
  - 테스트 비용 절감: 실제 API 호출 시 발생하는 과금, 데이터 전송 비용 등을 절감할 수 있습니다.
  - 테스트 효율 증대: 테스트를 위해 복잡한 데이터를 수동으로 셋업하거나, 특정 상태(예: 에러 응답)를 재현하기 위해 여러 조건을 맞출 필요가 없어집니다.
  - 비즈니스 로직 테스트에 집중: 외부 API 통신 로직이 아닌, API 응답을 기반으로 동작하는 순수한 비즈니스 로직을 검증하는 데 집중할 수 있습니다.
- 배치 테스트 적용
  - ItemReader 테스트: 배치 잡의 ItemReader가 외부 API에서 데이터를 읽어오는 경우, MockServer를 사용하여 다양한 API 응답 시나리오 (정상 응답, 빈 응답, 에러 응답 등)를 시뮬레이션하고 테스트할 수 있습니다.

MockRestServiceServer 사용

- MockRestServiceServer 란?
  - MockRestServiceServer는 RestTemplate 기반의 클라이언트 요청을 가로채고 **mock 응답**을 반환하는 Spring Test의 핵심 유틸리티입니다.
  - 이 도구는 실제 외부 인프라 네트워크 통신 없이 테스트를 진행할 수 있게 해줍니다.
- 동작 원리: 실제 서버처럼 작동하는 가짜 서버
  - **요청 가로채기:** MockRestServiceServer는 RestTemplate의 내부를 가로채어, 클라이언트가 보낸 요청이 실제 외부 서버로 나가지 못하게 막습니다.
  - **응답 정의:** 개발자는 테스트 코드 내에서 MockRestServiceServer에게 "이런 URL로 이런 요청이 오면, 이렇게 생긴 응답을 내려줘"라고 미리 설정합니다.
  - **가짜 응답 반환:** RestTemplate이 요청을 보내면, MockRestServiceServer는 실제 네트워크 통신 없이 로컬에서 미리 정의된 가짜 응답을 즉시 반환합니다.
  - 이러한 방식을 통해, 우리는 실제 서버의 상태나 네트워크 환경에 구애받지 않고 안정적이고 예측 가능한 테스트를 작성할 수 있습니다.

MockRestServiceServer를 활용한 테스트 코드 작성

```
public class SampleTest {
    @Autowired
    private RestTemplate restTemplate;
    private MockRestServiceServer mockServer;

    @Test
    public void testApiCallSuccess() {
        // 1. Given: MockServer에 응답을 정의
        mockServer.expect(requestTo("/api/data"))
                .andExpect(method(HttpMethod.GET))
                .andRespond(withSuccess("{\"key\":\"value\"}", MediaType.APPLICATION_JSON));

        // 2. When: 실제 RestTemplate을 사용하여 서비스 호출
        String result = restTemplate.getForObject("/api/…", String.class);

        // 3. Then: 결과 검증 및 MockServer 검증
        mockServer.verify(); // 예상한 요청이 정확히 이루어졌는지 확인
        //... 결과에 대한 추가적인 검증 로직
    }
}
```

MockServer 테스트의 단점

- 테스트 환경 설정의 복잡성
  - **MockServer는 별도의 서버를 로컬에 띄워**야 하는 추가적인 설정 과정이 필요합니다. 이는 테스트 환경을 복잡하게 만들고, 테스트 실행 시간을 늘릴 수 있습니다.
  - 특히 여러 API를 테스트하거나, 동적인 URL을 사용하는 경우 각 엔드포인트에 대한 MockServer 설정을 일일이 정의해야 하는 번거로움이 있습니다.
- 테스트의 과도한 종속성
  - MockServer를 사용하는 테스트 코드는 **"어떤 요청에 대해 어떤 응답을 줄 것인가"**를 구체적으로 명시해야 합니다. 이는 테스트 코드가 외부 API의 스펙에 과도하게 종속되게 만듭니다.
  - 만약 외부 API의 응답 스펙이 변경되면, MockServer의 응답 정의 코드를 모두 수정해야 합니다.
- 행위 기반(Behavior-based) 테스트의 어려움
  - MockServer는 결과 기반(Result-based) 테스트에 적합합니다. 즉, "이 요청을 보내면 이런 응답이 온다"를 검증하는 데 효과적입니다.
  - 하지만 Mockito 같은 프레임워크를 사용하는 행위 기반(Behavior-based) 테스트("이 메서드가 호출되었는가?")는 MockServer로 직접 구현하기 어렵습니다.

Mocking 기반 테스트 방법

- 행위 기반 Mocking이란?
  - 행위 기반 Mocking은 실제 통신이 아닌 메서드 호출 자체의 행위를 가짜(Mock)로 만드는 방법입니다. 즉, "이런 요청이 오면 이런 응답을 내려줘"라는 결과 기반의 방식이 아니라, "이 메서드가 호출되면 이런 값을 반환해줘"라고 정의하는 행위 기반의 방식입니다.
  - 이러한 방식을 사용하면, 우리는 실제 네트워크 통신 없이도 마치 통신이 성공한 것처럼 테스트를 진행할 수 있습니다. 가장 대표적인 라이브러리는 바로 Mockito입니다.
- 행위 기반 Mocking의 장점
  - 테스트의 간결함: 복잡한 네트워크 환경이나 MockServer 설정 없이, 필요한 클래스의 메서드 호출만 Mocking하여 테스트 코드를 매우 간결하게 만들 수 있습니다.
  - 유지보수 용이성: 외부 API의 스펙이 변경되더라도, 테스트 코드에서는 Mocking하는 메서드의 반환 타입만 수정하면 되므로 MockServer보다 유지보수 비용이 낮습니다.
  - 비즈니스 로직에 집중: 테스트의 관심사를 외부 인프라 통신에서 벗어나, 순수한 비즈니스 로직과 객체 간의 상호작용에만 집중하게 만듭니다.

Mockito 를 활용한 테스트 코드 작성

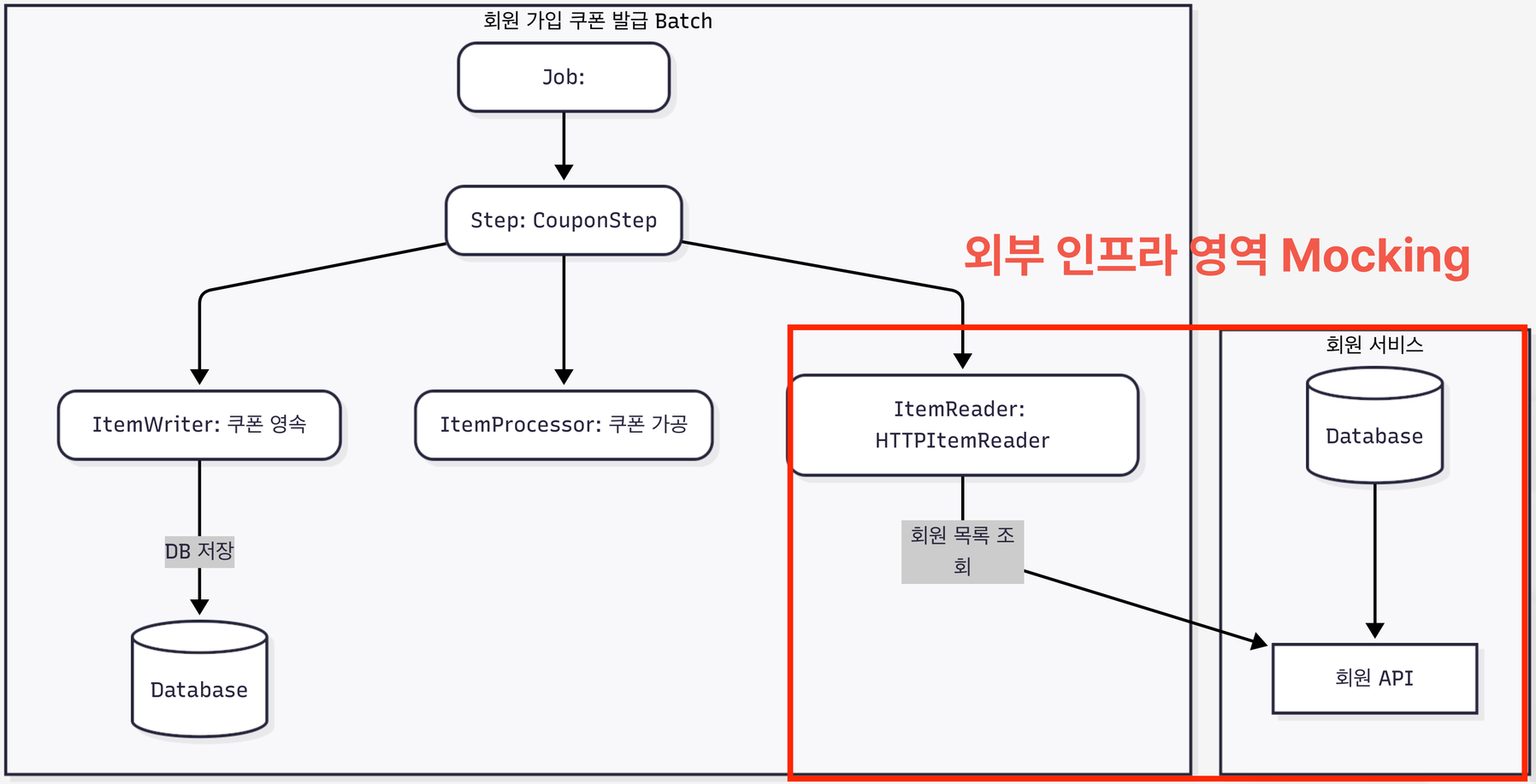
```
given(
        mockRestTemplate.exchange(
                queryParam.toUriString(),
                HttpMethod.GET,
                null,
                new ParameterizedTypeReference<PageResponse<MemberResponse>>() {
                }
        )
).willReturn(ResponseEntity.ok(responsePage0));

given(
        mockRestTemplate.exchange(
                uriBuilderPage1.toUriString(),
                HttpMethod.GET,
                null,
                new ParameterizedTypeReference<PageResponse<MemberResponse>>() {
                }
        )
).willReturn(ResponseEntity.ok(responsePage1));
```

Mocking 기반 테스트 단점

- 실제 통신 환경을 완벽히 모방할 수 없다.
  - Mocking은 실제 HTTP 통신을 하지 않는다는 근본적인 한계를 가집니다. 이는 네트워크 지연, 타임아웃, 예기치 않은 연결 오류 등 실제 환경에서 발생하는 문제를 테스트하기 어렵게 만듭니다.
  - 또한, 직렬화(Serialization) 및 역직렬화(Deserialization) 오류를 놓칠 수 있습니다. Mocking은 객체 메서드 호출을 가로채므로, 실제 JSON 또는 XML 변환 과정에서 발생할 수 있는 데이터 포맷 불일치나 필드명 오타와 같은 문제를 사전에 발견하기 어렵습니다.
- 테스트의 신뢰성이 떨어진다.
  - Mocking은 개발자가 정의한 가상의 시나리오에만 의존합니다. 실제 외부 API의 응답 데이터가 변경되거나, API 서버에 버그가 있는 경우에도 Mocking된 응답은 항상 동일하게 동작합니다. 이는 테스트는 성공하지만 실제 운영 환경에서는 실패하는 "테스트가 거짓말을 하는" 상황을 초래할 수 있습니다.
  - 이 때문에 Mocking 기반 테스트는 통합 테스트의 성격이 약해지는 단점이 있습니다. 순수한 비즈니스 로직을 검증하는 데는 탁월하지만, 외부 시스템과의 연동을 검증하는 데는 한계가 있습니다.

외부 인프라를 의존하는 테스트의 어려움 해결



외부 인프라를 의존하는 테스트의 어려움 해결

```
@TestConfiguration
public class CouponJobTestConfiguration {

    @Bean
    @Primary
    public RestTemplate mockRestTemplate() {
        return mock(RestTemplate.class);
    }
}
```

외부 인프라를 의존하는 테스트의 어려움 해결

```
@Test
void couponJobTest() throws Exception {
    // given
    // Mock for page 0
    final String pageResponse0 = readFile("/json/page-response-page-0.json");
    final PageResponse<MemberResponse> responsePage0 =
                    objectMapper.readValue(pageResponse0, new TypeReference<>() {});
    final UriComponentsBuilder queryParam0 = UriComponentsBuilder
                    .fromUriString("http://localhost:8080/api/v1/members")
            .queryParam("page", 0)
            .queryParam("size", 10);

    given(
            mockRestTemplate.exchange(
                    queryParam0.toUriString(),
                    HttpMethod.GET,
                    null,
                    new ParameterizedTypeReference<PageResponse<MemberResponse>>() {
                    }
            )
    ).willReturn(ResponseEntity.ok(responsePage0));

    // when
    launchJob(couponJob);
}
```

외부 인프라를 의존하는 테스트의 어려움 해결

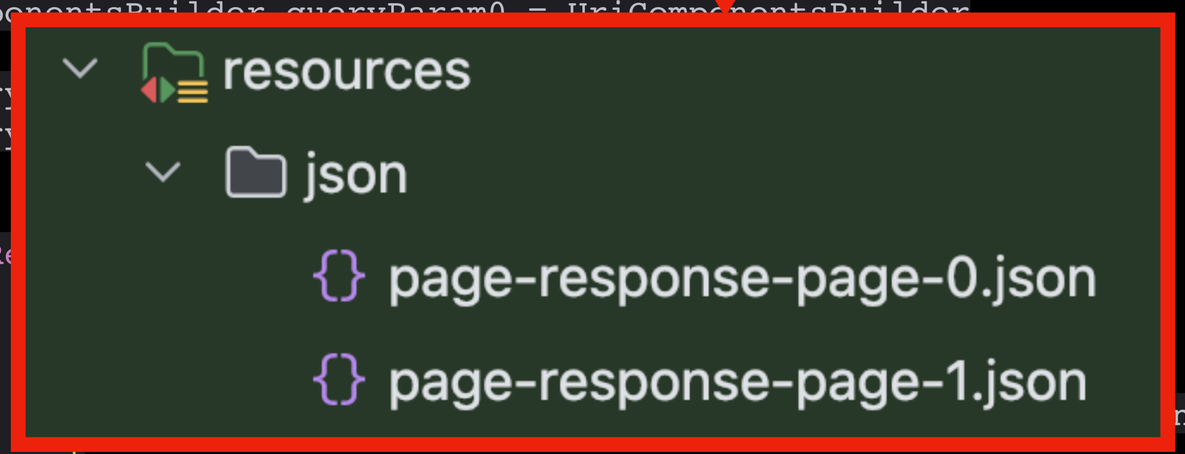
```
@Test
void couponJobTest() throws Exception {
    // given
    // Mock for page 0
    final String pageResponse0 = readFile("/json/page-response-page-0.json");
    final PageResponse<MemberResponse> responsePage0 =
                    objectMapper.readValue(pageResponse0, new TypeReference<>() {});
    final UriComponentsBuilder queryParam0 = UriComponentsBuilder

                .query
                .query

    given(
                mockRe

                        }
                )
    ).willReturn(ResponseEntity.ok(responsePage0));

    // when
    launchJob(couponJob);
}
```



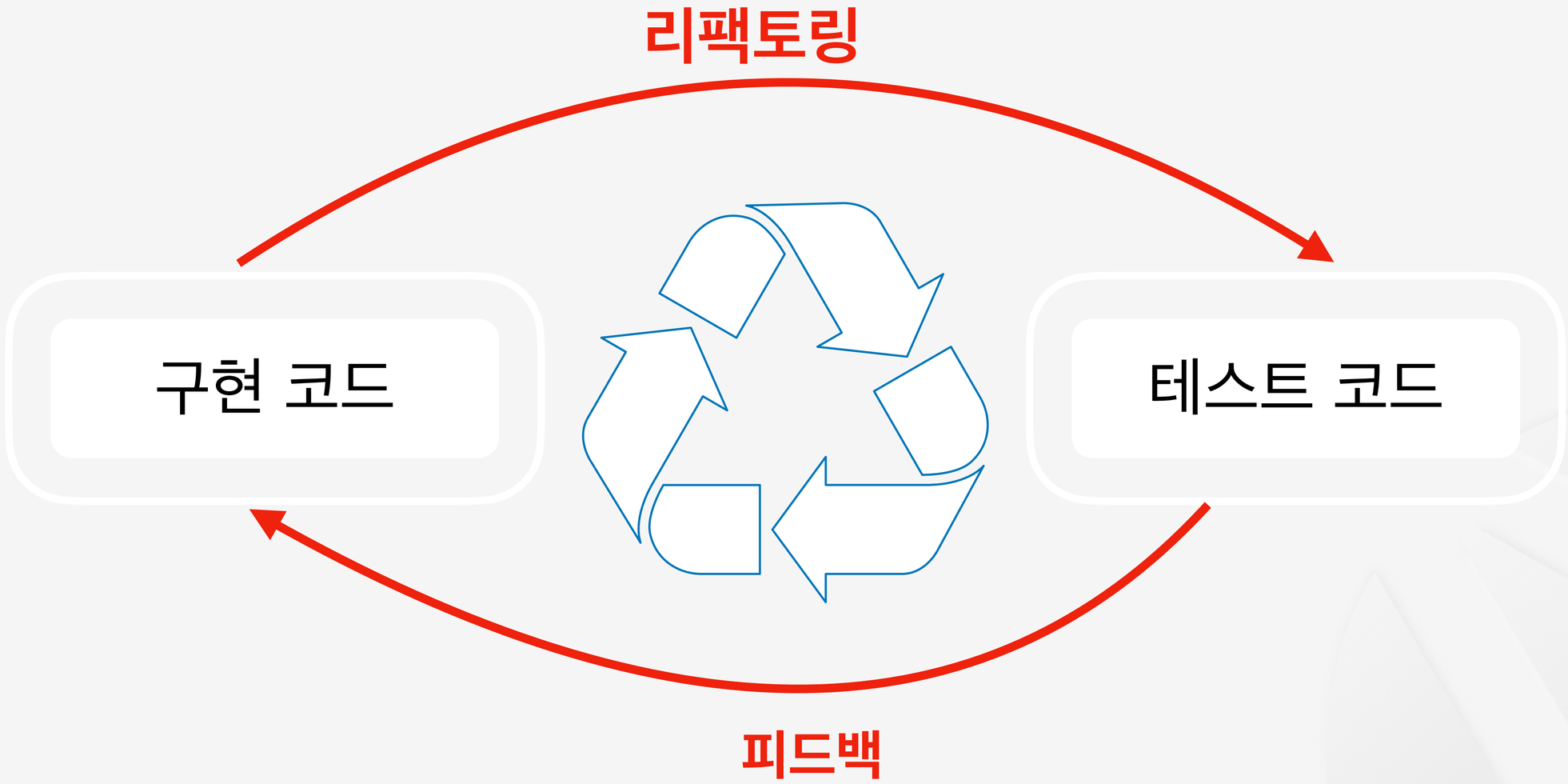
Part 3 Batch Test 작성하는 노하우 정리

- 배치 테스트의 효율적인 실행
  - JobLauncherTestUtils 활용: Spring Batch 테스트의 시작은 JobLauncherTestUtils입니다. 이 유틸리티를 사용하면 전체 Job 또는 특정 Step을 손쉽게 실행할 수 있습니다. launchJob()과 launchStep() 메서드를 통해 복잡한 Job을 효율적으로 테스트할 수 있습니다.
  - SpringBatchTestSupport 도입: 반복되는 테스트 설정을 개선하기 위해 SpringBatchTestSupport와 같은 추상 클래스를 만듭니다. 이 클래스에 @SpringBatchTest 같은 공통 어노테이션과 자주 사용하는 메서드들을 모아두면, 테스트 클래스는 상속만으로 모든 설정을 가져와 코드의 중복을 제거하고 유지보수성을 높일 수 있습니다.
- given 절의 효율적인 데이터 셋업
  - SQL, JSON 파일 기반 셋업: 복잡하고 귀찮은 테스트 데이터 셋업을 효율적으로 처리하기 위해 SQL이나 JSON 파일을 활용합니다.
  - SQL 파일: @Sql 어노테이션을 사용하여 테스트 전후로 SQL 스크립트를 실행함으로써, 복잡한 데이터베이스 관계를 코드 없이 관리할 수 있습니다. 이를 통해 테스트의 given 절이 간결해지고 다양한 케이스에 대한 테스트 데이터를 쉽게 확보할 수 있습니다.

Part 3 Batch Test 작성하는 노하우 정리

- 외부 인프라 의존성 해결
  - Mocking 기반 테스트: 배치 잡이 외부 API나 서비스에 의존하는 경우, Mocking을 통해 실제 통신 없이 테스트를 진행합니다.
  - Mockito 활용: Mockito와 같은 라이브러리를 사용해 외부 통신을 담당하는 객체의 메서드 호출을 가짜로 만듭니다. when().thenReturn()을 통해 특정 호출에 대한 가짜 응답을 정의함으로써, 실제 외부 환경에 의존하지 않고 테스트의 안정성을 확보할 수 있습니다.
  - MockRestServiceServer: RestTemplate을 사용하는 경우, MockRestServiceServer를 활용하여 로컬에서 가상의 서버를 띄워 HTTP 요청을 가로채고 가짜 응답을 반환하도록 할 수 있습니다.

지속 가능한 테스트 코드 작성 방법에 대한 고찰



지속 가능한 테스트 코드 작성 방법에 대한 고찰



END