패스트 캠퍼스 포인트 만료하기

4 프로젝트 업그레이드

프로젝트 생성하기

프로젝트 업그레이드

패스트 캠퍼스 포인트 관리하기 프로젝트를 만들기 위해서…
우리는 아래와 같은 다양한 일들을 했습니다.

1. 전반적인 포인트 시스템과 기획을 이해해봤습니다.
2. 포인트 관리하기 요구사항을 이해했습니다.
3. 개발 Task 도출
4. 프로젝트를 생성해보고 배치 프로젝트를 만들기 위한 과정들을 세팅해봤습니다.
    1. MySQL (docker)
    2. Gradle
    3. BatchTestSupport
5. 프로젝트의 메인 로직을 위한 테스트코드를 작성해보고 Job을 구현해봤습니다.
    1. 포인트 만료하기
    2. 포인트 예약 적립
    3. 포인트 만료 메시지 보내기
    4. 포인트 만료 예정 메시지 보내기

휴… 이젠 끝인가… → 꾸준히 유지보수하는 것도 하나의 업무

프로젝트 생성하기

문제점 1

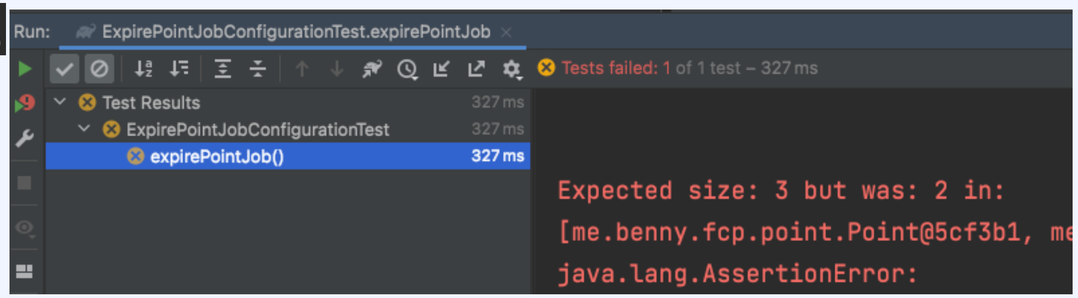
언제 발생할까?
두가지 조건은 다 만족하면 발생합니다.

1. 처리할 데이터 개수 > Page Size 인 경우
2. Reader의 조건으로 사용하는 필드와 Processor에서 수정하는 필드가 동일한 경우

reader processor

만료 여부 False인거 가져와야지! -> 만료 여부를 True로 바꿔야지

```
@Bean
@StepScope
public JpaPagingItemReader<Point> expirePointItemReader(
        EntityManagerFactory entityManagerFactory,
        @Value("#{T(java.time.LocalDate).parse(jobParameters[today])}")
                LocalDate today
) {
    return new JpaPagingItemReaderBuilder<Point>()
            .name("expirePointItemReader")
            .entityManagerFactory(entityManagerFactory)
            .queryString("select p from Point p where p.expireDate < :today and used = false and expired = false")
            .parameterValues(Map.of("today", today))
            .pageSize(1)
            .build();
}
```



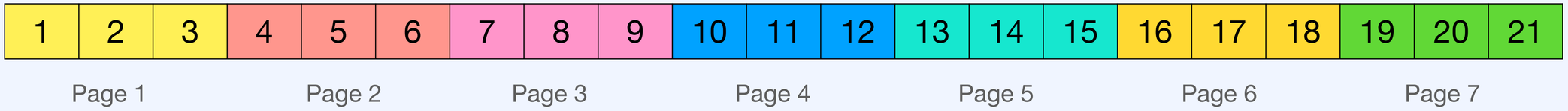
프로젝트 생성하기

문제점 1

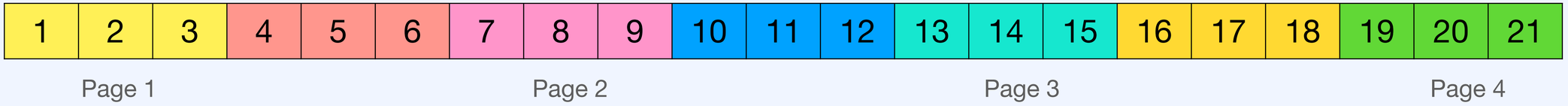
1. 처리할 데이터 개수 > Page Size 인 경우
2. Reader의 조건으로 사용하는 필드와 Processor에서 수정하는 필드가 동일한 경우

처리할 데이터 개수 21개
Page Size 3개

우리의 예상



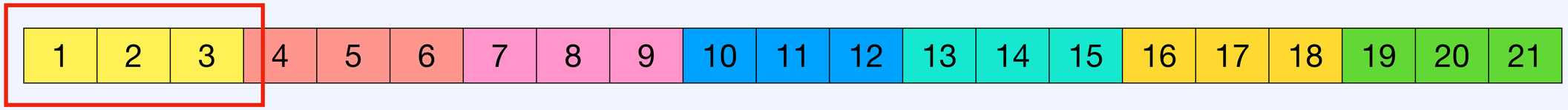
실제 결과



프로젝트 생성하기

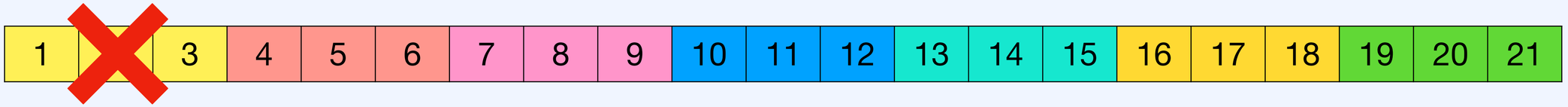
문제점 1

Page1 가져올때



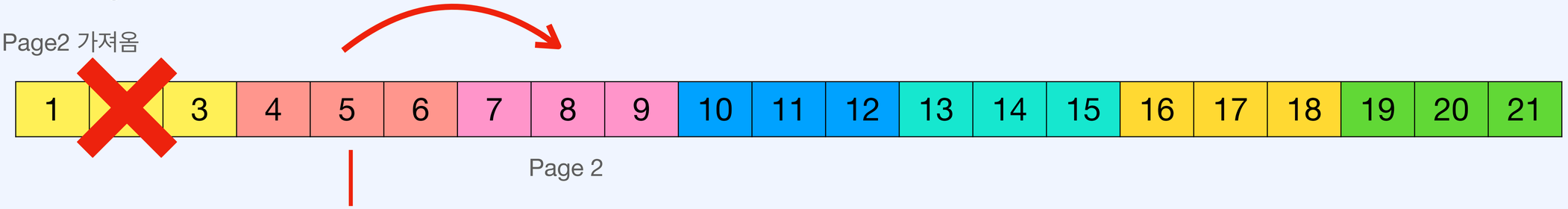
Page 1

Page1 가져온 것을 수정함 expired = true로 변경해버림 / 다시 검색하면 page1은 검색 되지 않음



Page 1

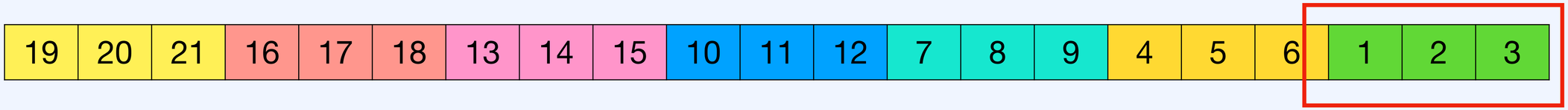
Page2 가져옴



프로젝트 생성하기

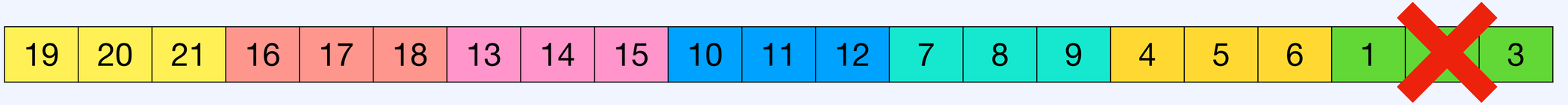
해결방법

Page1 가져올때



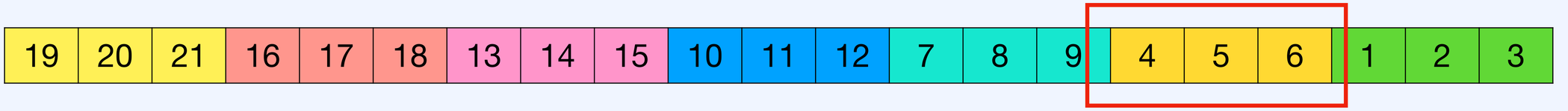
Page 1
= 마지막 페이지 - 0

Page1 가져온 것을 수정함 expired = true로 변경해버림 / 다시 검색하면 page1은 검색 되지 않음



Page 1

Page2 가져옴



Page 2
= 마지막 페이지 - 1

프로젝트 생성하기

문제점 2

우리 프로젝트는 아직 대량 처리 테스트, 성능 측정을 하지 않았습니다.
이것을 실제 운영에 배포한다고 생각해보시면 정말 안전하고 원하는대로 동작한다고 보장할 수 있을까요?

우리가 만든 배치 프로그램은 특정 시간에 서버를 구동하고 작업을 처리하고자 할 겁니다.
그런데 언제 시작시작할 지만 생각하고 언제 끝날지는 생각하지 않아도 될까요?

사업의 예측과 개발자의 목표로 하루에 만료처리할 포인트가 10만개이고 이것을 1시간 이내에 끝내겠다면
실제 10만개 처리에 걸리는 시간을 측정해야 하고 원하는 성능이 나오지 않으면 이것을 개선하기 위해 노력해야합니다.

우리가 만든 프로젝트를 계속해서 모니터링하고 문제가 있다면 꾸준히 개선해 나가야합니다.

1시간에 10만개? 별거 아닌데요?

현실은...?

1시간 = 3,600,000ms

데이터 개수 = 100,000개

흠....

데이터 1개 처리하는데 걸리는 시간 : 36ms

ㅠㅠ

그럼 100만개를 1시간에 처리하고 싶다면...?
개당 3.6ms만에 처리해야한다고!!?

프로젝트 생성하기

해결방법

배치 대량 처리 테스트해보기

실제로 배치 대량 처리 테스트를 통해 프로젝트의 안전성을 확보해야합니다.
예를 들면
우리 프로젝트에서 고객에게 유효기간 만료 메시지를 전송하는데 1시간이 걸릴수도 있고 12시간이 걸릴 수도 있습니다.
대량의 데이터를 넣어놓고 실제로 처리하는 과정을 해보겠습니다.

쿼리 튜닝하기

우리가 1장에서 봤었던 real-time 서비스나 stream-processing 서비스도 마찬가지로 사용하는 쿼리에 대해 튜닝이 필요합니다.
그러나 스프링 배치의 경우에는 유독 쿼리 튜닝이 필요합니다.
그 이유는 스프링 배치는 조회 성능에 따라 유독 크게 성능이 차이가 납니다.
튜닝된 select 쿼리를 사용하면 그렇지 않은 쿼리보다 수천배정도 차이가 납니다. (반대로 말하면 튜닝하지 않은 쿼리는 사용할 수도 없습니다.)

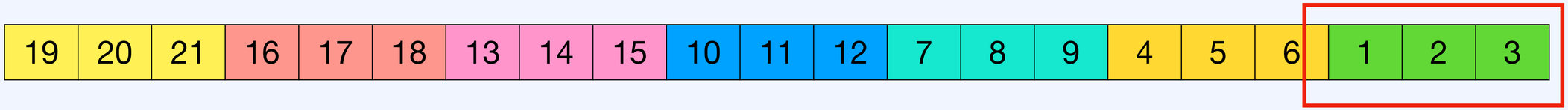
파티셔닝 하기

현재 우리 프로젝트는 싱글 스레드로 데이터를 처리합니다. (대부분의 배치프로그램이 그렇긴합니다)
하지만 병렬로 데이터를 처리한다면 처리 속도가 증가될 것입니다.
위의 2가지 케이스를 모두 다루고 그래도 성능을 개선하고 싶다면 파티셔닝을 할 수 있습니다.
병렬 처리라는 것은 성능 개선에 도움을 주지만 동시성 처리를 해야하기 때문에 함부로 사용할 수 없고 에러처리와 동작을 이해하는데 어려움이 생깁니다.
그렇기 때문에 위의 두가지를 먼저 시도해보고 그럼에도 개선점이 없어보인다면 파티셔닝을 시도하는 것이 좋습니다.

ReverseJpaPagingItemReader 구현하기

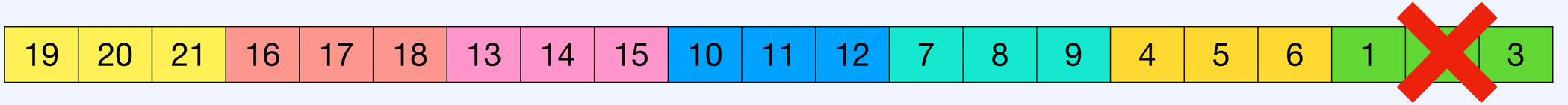
저번에 보았던 해결방법

Page1 가져올때



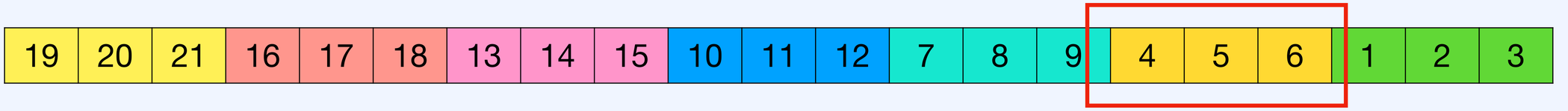
Page 1
= 마지막 페이지 - 0

Page1 가져온 것을 수정함 expired = true로 변경해버림 / 다시 검색하면 page1은 검색 되지 않음



Page 1

Page2 가져옴



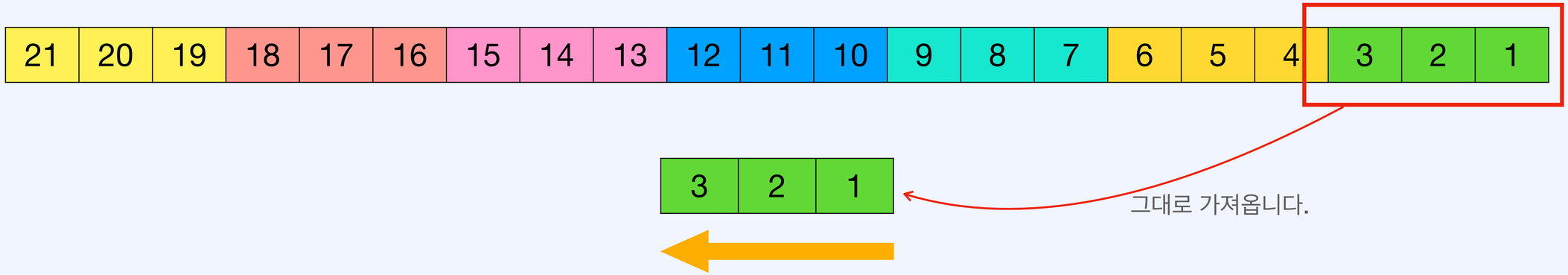
Page 2
= 마지막 페이지 - 1

ReverseJpaPagingItemReader 구현하기

그럼 실제로 어떻게하면 저렇게 뒤집을 수 있을까?

먼저 전체 순서를 거꾸로 나열합니다.

가장 뒤의 페이지부터 읽습니다.



ReverseJpaPagingItemReader 구현하기

새로운 ItemReader 만들기

이렇게 발생한 문제는 read할때 발생하는 문제입니다.

따라서 위의 요구사항을 충족하는 새로운 ItemReader가 필요합니다.

ReverseJpaPagingItemReader를 직접 구현해보도록 하겠습니다.

기존 프로젝트에서 브랜치를 upgrade로 바꾸시면 결과물을 확인할 수 있습니다.
https://github.com/kker5/point-management-practice/tree/upgrade

ReverseJpaPagingItemReader 구현하기

새로운 ItemReader 적용하기

기존코드

```
@Bean
@StepScope
public JpaPagingItemReader<Point> expirePointItemReader(
        EntityManagerFactory entityManagerFactory,
        @Value("#{T(java.time.LocalDate).parse(jobParameters[today])}")
                LocalDate today
) {
    return new JpaPagingItemReaderBuilder<Point>()
            .name("expirePointItemReader")
            .entityManagerFactory(entityManagerFactory)
            .queryString("select p from Point p where p.expireDate < :today and used = false and expired = false")
            .parameterValues(Map.of("today", today))
            .pageSize(1000)
            .build();
}
```

ReverseJpaPagingItemReader 구현하기

새로운 ItemReader 적용하기

수정된 코드

```
@Bean
@StepScope
public ReverseJpaPagingItemReader<Point> expirePointItemReader(
        PointRepository pointRepository,
        @Value("#{T(java.time.LocalDate).parse(jobParameters[today])}")
                LocalDate today
) {
    return new ReverseJpaPagingItemReaderBuilder<Point>()
            .name("messageExpireSoonPointItemReader")
            .query(
                    pageable -> pointRepository.findPointToExpire(today, pageable)
            )
            .pageSize(1000)
            .sort(Sort.by(Sort.Direction.ASC, "id"))
            .build();
}
```

```
Page<Point> findPointToExpire(LocalDate today, Pageable pageable);
```

```
Function<Pageable, Page<T>> query
```

Repository에 Page를 반환하는 메소드를
구현한 다음에 여기다 넣으면 됩니다.

Sorting하는 법
실제 내부에서는 id를 DESC로 나열하고 뒤에서부터 데이터를 읽어서
순서에는 변함이 없습니다.

ReverseJpaPagingItemReader 구현하기

새로운 ItemReader 만들기

```
public void setSort(Sort sort) {
    if (!Objects.isNull(sort)) {
        // pagination을 마지막 페이지부터 하기때문에 sort direction를 모두 reverse한다.
        // ASC <-> DESC
        Iterator<Sort.Order> orderIterator = sort.iterator();
        final List<Sort.Order> reverseOrders = Lists.newLinkedList();
        while (orderIterator.hasNext()) {
            Sort.Order prev = orderIterator.next();
            reverseOrders.add(
                    new Sort.Order(
                            prev.getDirection().isAscending() ? Sort.Direction.DESC : Sort.Direction.ASC,
                            prev.getProperty()
                    )
            );
        }
        this.sort = Sort.by(reverseOrders);
    }
}
```

Sort를 가져와서 순서를 역순으로 바꿉니다.
ASC -> DESC

ReverseJpaPagingItemReader 구현하기

새로운 ItemReader 만들기

Step 전에 실행합니다.
스텝 전에 1번만 실행된다고 보면됩니다.

```
@BeforeStep
public void beforeStep() {
    // 우리는 뒤에서부터 읽을 것이기 때문에 마지막 페이지 번호를 구해서 page에 넣어줍니다.
    totalPage = getTargetData(0).getTotalPages();
    page = totalPage - 1;
}
```

먼저 전체 페이지 개수를 구해서 저장합니다.

```
/**
 * page 번호에 해당하는 데이터를 가져와서 Page 형식으로 반환합니다.
 */
private Page<T> getTargetData(int readPage) {
    return Objects.isNull(query)?Page.empty():query.apply(PageRequest.of(readPage, pageSize, sort));
}
```

```
@BeforeRead
public void beforeRead() {
    if (page < 0)
        return;
    if (readRows.isEmpty()) // 읽은 데이터를 모두 소진하면 db로 부터 데이터를 가져와서 채워놓는다.
        readRows = Lists.newArrayList(getTargetData(page).getContent());
}
```

read()를 하기전에 데이터가 비어있으면 데이터를 가져와서 채워줍니다.
page는 가장 마지막 번호부터 가져옵니다.

ReverseJpaPagingItemReader 구현하기

새로운 ItemReader 만들기

```
@Override
public T read() {
    // null을 반환하면 Reader는 모든 데이터를 소진한걸로 인지하고 종료합니다.
    // 데이터를 리스트에서 거꾸로 (readRows.size() - 1) 뒤에서부터 빼줍니다.
    return readRows.isEmpty() ? null : readRows.remove(readRows.size() - 1);
}
```

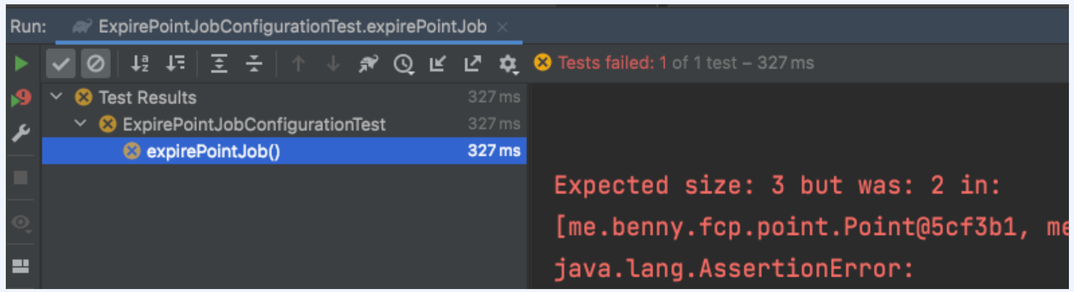
beforeRead에서 채워준 데이터(readRows)에서 가장 뒷쪽부터 하나씩 빼서 반환합니다.

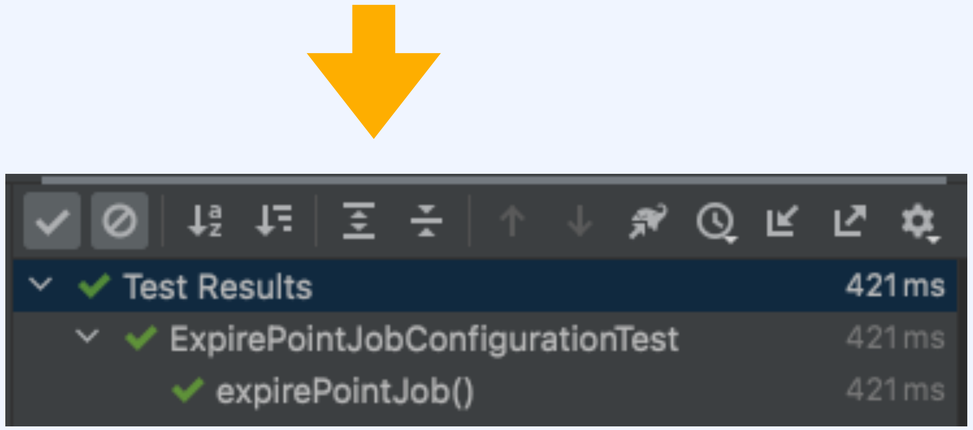
```
@AfterRead
public void afterRead() {
    // 데이터가 없다면 page를 1 차감합니다.
    if (readRows.isEmpty()) {
        this.page--;
    }
}
```

한 페이지를 다 사용하면 page를 1 차감합니다.

ReverseJpaPagingItemReader 구현하기

ReverseJpaPagingItemReader를 사용한 결과





데이터 대량처리 테스트

일단 대량으로 데이터를 넣어보자

```
docker exec -i -t <container Id> /bin/bash
mysql -u root -p
use point;
```

일단 포인트 지갑 2개 생성

user1, user2 의 지갑생성

```
INSERT INTO `point_wallet`
(user_id, amount)
VALUES ('user1', 0);
INSERT INTO `point_wallet`
(user_id, amount)
VALUES ('user2', 0);
```

포인트 예약적립 건 데이터 대량 insert 프로시저 생성

프로시저를 사용하지 않고 그 어떤 방법으로도 데이터를 대량으로 만들수만 있으면 상관없습니다.

```
DELIMITER //
CREATE PROCEDURE bulkInsert()
BEGIN
DECLARE i INT DEFAULT 1;
DECLARE random_amount bigint DEFAULT 1;
WHILE (i <= 10000) DO
SET random_amount = FLOOR(1000 + (RAND() * 9000));
INSERT INTO `point_reservation`
(point_wallet_id, amount, earned_date, available_days, is_executed)
VALUES (1, random_amount, '2021-09-13', 10, 0);
SET i = i+1;
END WHILE;
END;
//
DELIMITER ;
```

10000개를 insert합니다.

1000원 부터 10000원까지 랜덤금액을 넣음

point_wallet Id와 적립일자확인 필수!

프로시저 실행하는 방법

call bulkInsert();

데이터 대량처리 테스트

일단 대량으로 데이터를 넣어보자

대량으로 넣은 데이터를 확인해보자

call bulkInput(); 을 할때마다 1만개의 데이터가 insert 됩니다.

```
mysql> call bulkInsert();
Query OK, 1 row affected (15.54 sec)

mysql> select count(1) from point_reservation;
+----------+
| count(1) |
+----------+
|    20000 |
+----------+
1 row in set (0.09 sec)
```

두번 돌렸더니 20000개 생성

bulkInsert 프로시저 삭제하는 방법

drop procedure if exists bulkInsert;

데이터 대량처리 테스트

일단 대량으로 데이터를 넣어보자

```
DELIMITER //
CREATE PROCEDURE bulkNoTarget()
BEGIN
DECLARE i INT DEFAULT 1;
DECLARE random_amount bigint DEFAULT 1;
WHILE (i <= 100000) DO
SET random_amount = FLOOR(1000 + (RAND() * 9000));
INSERT INTO `point_reservation`
(point_wallet_id, amount, earned_date, available_days, is_executed)
VALUES (1, random_amount, '2030-12-30', 10, 0);
SET i = i+1;
END WHILE;
END;
//
DELIMITER ;
```

오늘 적립 대상이 아닌 것들도 넣어보자

데이터 대량처리 테스트

100만개의 데이터를 넣고 확인해보자

```
mysql> select count(1) from point_reservation;
+----------+
| count(1) |
+----------+
|  1000000 |
+----------+
1 row in set (0.19 sec)
```

전체 데이터 개수 100만개

```
mysql> select count(1) from point_reservation where is_executed=0 and earned_date = '2021-09-13' ;
+----------+
| count(1) |
+----------+
|   200000 |
+----------+
1 row in set (1.29 sec)
```

9월 13일 적립 대상 20만개

데이터 대량처리 테스트

100만개의 데이터를 넣고 확인해보자

배치 스트레스 테스트 변수 파악하기

Docker MySQL

실제 운영 환경에서는 도커 MySQL을 사용하지 않을 것이기 때문에 환경이 다릅니다.
Docker MySQL은 적은 리소스로 동작하기 때문에 실제 운영환경과 같은 환경에서의 테스트라고 보기 어렵습니다.

네트워크

Docker MySQL은 네트워크 비용이 거의 들지 않습니다.
바로 로컬에 있는 Database에 접근하여 데이터를 가져오고 쓰기 때문입니다.
이 또한, 실제 운영환경과 같은 환경에서의 테스트라고 보기 어렵습니다.

데이터 개수

사업이 얼마나 잘되고 있는지에 따라 다르겠지만 애초에 100만개라는 데이터가 많은 데이터는 아닙니다.

데이터 분포도

데이터 분포도 즉, 카티널리티가 운영환경과 스트레스테스트 환경이 다릅니다.
운영 환경에서는 훨씬 더 다양한 케이스 별로 데이터가 분포되어있을 것입니다.
스트레스 테스트 환경에서는 그런 다양한 케이스를 만들기가 어렵습니다. (하나하나 넣어주는게 힘듭니다.)

데이터 대량처리 테스트

100만개의 데이터를 넣고 확인해보자

20만개 데이터 예약건 적립하기

실행

```
java -jar point-management-practice-1.0-SNAPSHOT.jar --job.name=executePointReservationJob today=2021-09-13
```

결과

약 30분 소요

상황에 따라 20만개 처리하는데 30분이라면 만족할 수도 있습니다.
그러나 100만개 처리를 목표로 두고 있는 상황이라면 150분 (약 4시간)이 소요되므로 만족할만한 수치는 아닐겁니다.

쿼리 튜닝하기

쿼리 튜닝이 필요한 이유

조회성능 = 배치성능?

배치 성능에 가장 큰 영향을 미치는 것 중에 하나가 바로 ItemReader의 조회성능입니다.
프로젝트마다 특징이 다르기 때문에 반드시 그런것은 아니지만
대부분의 경우에는 ItemReader에서 가져온 데이터를 수정하거나 변환해서 데이터베이스에 저장합니다.
수정하거나 변환하는 부분, 그리고 데이터베이스에 저장하는 부분은 눈에 띄는 개선이 있기 어려운 부분입니다.
(물론 JPA를 쓰지 않고 다른 방식을 사용한다면 크게 개선될 수 있습니다.)
하지만 Reader의 조회 쿼리 성능은 배치 성능에 매우 큰 영향을 줍니다.

특히나, 만료 메시지와 만료예정 메시지와 같이 group by하고 sum
하는 복잡한 쿼리의 경우에는 조회 성능이 압도적으로 중요합니다.

```
select
w.user_id,
sum(p.amount)
from point p
inner join point_wallet w
on p.point_wallet_id = w.id
where p.is_expired = 1
and p.is_used = 0
and p.expire_date = ‘2021-01-01’
group by p.point_wallet_id
order by p.point_wallet_id asc
limit 1000, 0;
```

쿼리 튜닝하기

쿼리 튜닝 어떻게 하나요?

데이터의 분포도에 따라 쿼리 조회 결과가 다르지만 95만개 정도의 데이터중에 조회하고자 하는 데이터를 4천개 만들어두었습니다.

전체 데이터는 95만건

```
mysql> select count(1) from point_reservation;
+----------+
| count(1) |
+----------+
|   945895 |
+----------+
1 row in set (0.14 sec)
```

적립 예정 건은 4000개

```
mysql> select count(1) from point_reservation where is_executed=0 and earned_date = '2021-09-13' ;
+----------+
| count(1) |
+----------+
|     4000 |
+----------+
```

1000개 가져오는데 1.6초나 걸리다니!

```
1000 rows in set (1.59 sec)

mysql> select * from point_reservation where is_executed=0 and earned_date = '2021-09-13' limit 3000, 1000;
```

쿼리 튜닝하기

쿼리 튜닝 어떻게 하나요?

쿼리를 튜닝하기 위해서는 explain을 통한 확인이 필요합니다.

```
mysql> explain select * from point_reservation where is_executed=0 and earned_date = '2021-09-13' limit 3000, 1000;
+----+-------------+-------------------+------------+------+---------------+------+---------+------+--------+----------+-------------+
| id | select_type | table             | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra       |
+----+-------------+-------------------+------------+------+---------------+------+---------+------+--------+----------+-------------+
|  1 | SIMPLE      | point_reservation | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 943179 |     1.00 | Using where |
+----+-------------+-------------------+------------+------+---------------+------+---------+------+--------+----------+-------------+
1 row in set, 1 warning (0.03 sec)
```

처음부터 모든 데이터를 검색하는 구나... 그러니까 느리지... 인덱스가 필요해!

alter table point_reservation add index idx_executed_earndate(is_executed, earned_date);

```
mysql> explain select * from point_reservation where is_executed=0 and earned_date = '2021-09-13' limit 3000, 1000;
+----+-------------+-------------------+------------+------+-----------------------+-----------------------+---------+-------------+------+----------+-------+
| id | select_type | table             | partitions | type | possible_keys         | key                   | key_len | ref         | rows | filtered | Extra |
+----+-------------+-------------------+------------+------+-----------------------+-----------------------+---------+-------------+------+----------+-------+
|  1 | SIMPLE      | point_reservation | NULL       | ref  | idx_executed_earndate | idx_executed_earndate | 4       | const,const | 4000 |   100.00 | NULL  |
+----+-------------+-------------------+------------+------+-----------------------+-----------------------+---------+-------------+------+----------+-------+
1 row in set, 1 warning (0.02 sec)
```

추가한 인덱스를 사용해서 쿼리가 동작했다!

```
1000 rows in set (0.09 sec)

mysql> select * from point_reservation where is_executed=0 and earned_date = '2021-09-13' limit 3000, 1000;
```

1.6초에서 0.09초로 속도 상승!

쿼리 튜닝하기

쿼리 튜닝 주의할 점

Index 마구 추가 하지 않기

Index가 많아지면 그 테이블에 데이터를 Insert할 때 시간이 증가됩니다.

인덱스를 마구 추가하게 되면 원래 계획했던 대로 쿼리가 동작하지 않을 수도 있습니다.
(결과는 동일하겠지만 빠른 방법을 두고 굳이 어려운 방법으로 동작할 수도 있습니다.)

데이터 분포도에 따라 의미있던 Index가 의미가 없어지기도 합니다.
서비스가 진행됨에 따라 데이터들의 분포도가 바뀔수 있기 때문입니다.
따라서 섣불리 인덱스를 추가하지 않고 꼼꼼하게 확인하거나 DB전문가의 도움을 받는게 좋습니다.

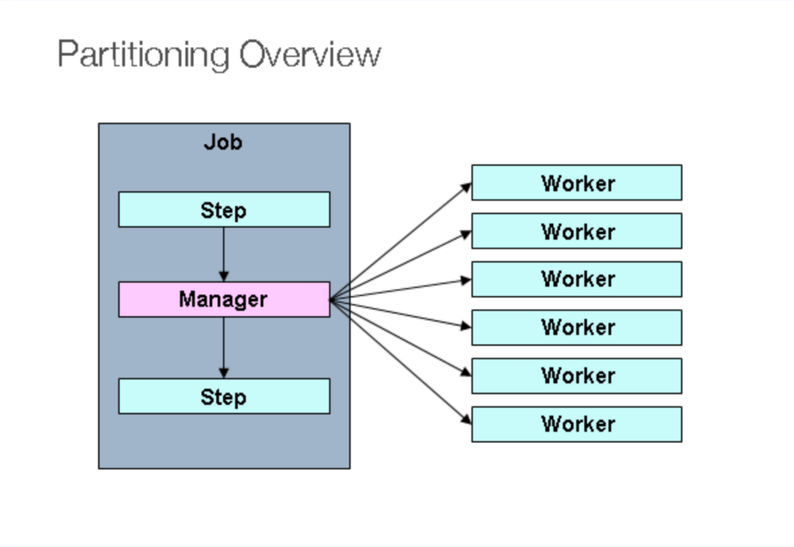
Index 추가 보다 조회 쿼리 자체를 수정해보기

인덱스로 모든 것을 해결하려고 하기 보다는 쿼리 자체를 수정하는 것이 더 효율적인 경우가 많습니다.
예를 들면 orderby 대상을 변경해본다던가 또는 join을 subquery로 변경해본다던가
이런 쿼리 수정을 통해서 성능 개선을 이뤄낼수도 있습니다.

파티셔닝

파티셔닝이란

1. Master Step을 이용해 처리할 데이터를 더 작게 나눕니다.
2. Worker Step을 통해 나눈 데이터를 처리하도록 합니다.
3. Local에서 처리하면 Multi Thread로 처리합니다.
4. Remote로 처리하면 여러 서버가 동시에 처리합니다.



파티셔닝

파티셔닝 구현하는 방법

Partitioner와 PartitionHandler를 구현해야합니다. https://github.com/kker5/point-management-practice/tree/partition

Partitioner를 구현

```
public interface Partitioner {

    Map<String, ExecutionContext> partition(int gridSize);

}
```

파티션이름 & StepExecutionContext 세트

StepExecution 생성 개수

PartitionHandler를 구현

Master 스텝이 Worker 스텝을 어떻게 관리하는 지를 결정하는 부분입니다.
크게 두가지 중에 하나를 사용합니다.

- TaskExecutorPartitionHandler
  로컬에서 Thread로 분할
- MessageChannelPartitionHandler
  원격으로 메타 데이터 전송

파티셔닝

파티셔닝 구현 코드

파티셔너 구현

```
@RequiredArgsConstructor
public class ExecutePointReservationStepPartitioner implements Partitioner {
    private final PointReservationRepository pointReservationRepository;
    private final LocalDate today;
    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {
        long min = pointReservationRepository.findMinId(today);
        long max = pointReservationRepository.findMaxId(today);
        long targetSize = (max - min) / gridSize + 1;
       while (start <= max) {
            ExecutionContext value = new ExecutionContext();
            value.putLong("minId", start);
            value.putLong("maxId", end);
            result.put("partition" + number, value);
        }
        return result;
    }
}
```

오늘 처리할 포인트 예약건의 최소 Id와 최대Id를 구함

gridSize 로 Id를 분할합니다.
gridSize: 4 / min : 1 / max : 20
이면 아래와 같이 분할됩니다.
1~5 / 6~10 / 11~15 / 16~20

minId와 maxId를 StepExectuionContext에 넣음

파티션이름, StepExecutionContext를 반환 데이터에 넣는다.

파티셔닝

파티셔닝 구현 코드

로컬에서 Thread로 파티셔닝 진행

```
@Bean
public TaskExecutorPartitionHandler partitionHandler(
        Step executePointReservationStep,
        TaskExecutor taskExecutor
) {
    TaskExecutorPartitionHandler partitionHandler = new TaskExecutorPartitionHandler();
    partitionHandler.setStep(executePointReservationStep);
    partitionHandler.setGridSize(4);
    partitionHandler.setTaskExecutor(taskExecutor);
    return partitionHandler;
}
```

grid size

파티셔닝

파티셔닝 구현 코드

```
@Bean
@JobScope
public Step executePointReservationMasterStep(
        StepBuilderFactory stepBuilderFactory,
        TaskExecutorPartitionHandler partitionHandler,
        PointReservationRepository pointReservationRepository,
        @Value("#{T(java.time.LocalDate).parse(jobParameters[today])}")
                LocalDate today
) {
    return stepBuilderFactory
            .get("executePointReservationMasterStep")
            .partitioner(
                    "executePointReservationStep",
                    new ExecutePointReservationStepPartitioner(pointReservationRepository, today)
            )
            .partitionHandler(partitionHandler)
            .build();
}
```

최종적으로는…
4개의 Thread가 WorkerStep을 만들어
처리해야할 데이터를 (ID기준으로) 4등분해서
WorkerStep이 각자 데이터를 처리함

Worker Step이 될 Step 이름 추가

Partitioner 추가

PartitionHandler추가

파티셔닝

파티셔닝 구현 코드

```
@Bean
@StepScope
public ReverseJpaPagingItemReader<PointReservation> executePointReservationItemReader(
        PointReservationRepository pointReservationRepository,
        @Value("#{T(java.time.LocalDate).parse(jobParameters[today])}") LocalDate today,
        @Value("#{stepExecutionContext[minId]}") Long minId,
        @Value("#{stepExecutionContext[maxId]}") Long maxId
) {
    return new ReverseJpaPagingItemReaderBuilder<PointReservation>()
            .name("messageExpireSoonPointItemReader")
            .query(
                    pageable -> pointReservationRepository.findPointReservationToExecute(today, minId, maxId, pageable)
            )
            .pageSize(1000)
            .sort(Sort.by(Sort.Direction.ASC, "id"))
            .build();
}
```

StepExecutionContext로 minId, maxId 사용가능

minId, maxId를 추가해서 WorkerStep이 맡아야할 4등분 되어 감소함

파티셔닝

파티셔닝 주의 사항

파티셔닝 하기 전에…

배치 성능을 개선하려고 한다면 중요한 선택지 중에 하나가 바로 파티셔닝입니다.
상황만 잘 맞는다면 당연히 눈에 띄는 개선을 이룰수 있습니다.
그러나 파티셔닝을 하기에 앞서 자신의 코드와 코드에서 사용하는 쿼리를 개선해서 성능을 개선하는 것을 먼저 해보는 걸 추천합니다.
문제 있는 코드가 있다면 아무리 스레드를 늘린다고 하더라도 근본적으로 개선되기는 어렵습니다.

동시성 처리

파티셔닝을 하게되면 병렬처리를 합니다.
즉, 동시성 문제를 겪게 됩니다.
제가 본 배치 Job의 절반 이상은 이런 동시성 문제때문에 파티셔닝이 불가한 상태였습니다.
만약에 중간에 동시성 문제를 해결하기 위해 Lock을 걸게 되면 병렬처리는 하는 의미가 없어져 오히려 성능이 저하될 수도 있습니다.