# 01.15

# 피드백 시간

## 📌안정해시설계 - 수평적 규모 확장성 달성하기

- 안정적으로 데이터 균등 분배를 위해 사용되었다. 해시테이블 크기가 조정될때 평균적으로 K/n개의 키만 재배치하는 것.
    - 서버와 키를 균등분호 해시함수를 이용해 해시링에 배치
    - 키의 위치에서 링을 시계방향으로 탐색하다 만나는 최초의 서버가 키가 저장될 서버

**장점**

- 서버가 추가되거나 삭제될때 재배치되는 키의 수가 최소화됨
- 데이터가 보다 균등 분포( 수평적 규모 확장성 달성)

**단점**

- 파티션 크기 균등 유지 X
- 키의 균등 분포 X

극복 : **가상 노드 기법 도입** → 핫스팟 키문제 감소 - 서버 과부하 감소

하나의 서버가 여러개의 가상노드를 갖는것.

## 📌인덱스 ( 기초 )

추가적인 쓰기 작업과 저장 공간을 활용하여 데이터베이스 테이블의 검색 속도를 향상시키기 위한 자료구조

- B-Tree 형태이다.

root node, branch node, leaf node로 구성되어 있고, leaf node에는 디비의 주소값까지 가지고 있다.

- DBMS 인덱스는 항상 **정렬**되어 있기 때문에 빠르게 검색이 가능하다.

  데이터 파일은 레코드 삭재 후 빈공간 존재시 삭제된 공간을 재활용하기 때문에 정렬되어 있지 않다.

- 데이터의 저장( Insert, update, delete)성능을 희생하고 읽기 속도를 높이는 기능
  - delete(인덱스 키 삭제)

    삭제마크를 한다 → **삭제하는 데이터의 인덱스를 사용하지 않는다는 작업을 추가로 해야하기 때문에 성능저하가 발생한다.**

    인덱스 키 삭제로 인한 **마킹 작업 또한 디스크 쓰기가 필요하다.**

  - update(인덱스 키 변경)

    키값을 삭제후 새로운 키값이 추가된다.

  - insert(새로운 인덱스키 추가) :

    저장될 위치가 결정되면 리프노트에 추가되며, 리프노트가 다 차면 split 현상이 발생한다.

## 📌트랜잭션 격리레벨

1) READ UNCOMMITTED
-- 버퍼풀을 읽음 ( 새로 업데이트된 값을 무조건 읽음 )
-- 더티 리드 발생 ( 트랜잭션 처리전에도 업데이트 값을 읽게 됨 )

2) READ COMMITTED
-- 언두 로그를 읽음 ( 수정 전 데이터 읽음, 백업용 )
-- 오라클 DBMS에서 default
-- 같은 트랜잭션에도 select 쿼리 여러 번 실행 시 커밋 후 읽었지만 다른 결과가 나오는 경우 발생

3) REPEATABLE READ
-- 언두 로그를 읽음 ( 수정 전 데이터 읽음, 백업용 )
-- MySQL innoDB default
-- 트랜잭션마다 번호를 가져서(id), 자신의 번호보다 작은 번호의 트랜잭션에서 변경된 것만 언 두로그에서 볼 수 있음(현재 트랜잭션 이후의 트랜잭션의 변경사항을 볼 수 없음)
 읽는 쪽에서  쓰는 것을 잠금X, 쓰는 곳에서도 읽는 것 잠금X

-- phantom read 발생

4) SERIALIZABLE
-- 언두 로그를 읽음 ( 수정 전 데이터 읽음, 백업용 )
-- 갭락과 넥스트 키락으로 PHANTOM READ 방지
-- 읽기 작업도 읽기 잠금을 획득해야함. -> 다른 트랜잭션은 절대 접근 불가

## 📌지연 로딩이 적용 안되는 경우

프록시를 사용할 때 외래키(FK)를 직접 관리하지 않는 일대일 관계는 지연로딩으로 설정해도 즉시로딩이 된다.

@OneToOne에서 외래키가 A에 있으면 A를 조회하는 순간 B의 데이터가 있는지 확인할 수 있다. 그래서 null을 입력해야 할지, 아니면 프록시를 입력해야 할지 명확한 판단이 가능하다.

외래키가 없는 B를 조회할 때 A가 데이터가 있는지 없는지 판단이 불가능하다. 그래서 null을 입력할지 아니면 프록시를 입력해야 할지 판단히 불가능하다. 따라서 이 경우 강제로 즉시 로딩을 해서 데이터가 있으면 해당 데이터를 넣고, 없으면 null을 입력하게 된다.

- FK를 PK라고 말함.

## 📌UncheckedException vs checkedException

### UncheckedException

- 확인되지 않은 실수 (개발자의 실수).
- 문법적으로 uncheck라 정해놓았기 때문에, 컴파일 시점에 예외를 catch 하는지 확인하지 않는다.
- spring에서는 default가 트랜잭션 Rollback 이 되는 것이다.→ rollback여부 바꿀 수 있다.
- 관련 예외
    - RuntimeException - NullPointerException, IndexOutOfBoundException

### checkedException

- java문법에 의해서 check를 해야한다. 명시적으로 checked로 되어 있는 구문이다

- CheckedException 이 발생할 가능성이 있는 로직은 반드시 **복구, 회피, 처리**해줘야 한다.
- 컴파일 시점에서 예외에 대한 처리가 안되어있다면 with try/catch 컴파일 에러가 발생한다. 처리를 하지 않았다면 예외가 발생하는 메서드에서 throws 예약어를 활용해서 던져야 한다
- spring에서는 default가 트랜잭션 Rollback 이 안되게 되어있다. 즉,checked exception은 발생해도 트랜잭션이 commit 돼버리므로 주의해야한다 → rollback여부 바꿀 수 있다.
- 관련 예외
    - IOException, SQLException

**복구, 회피, 처리 방법**

**1) 복구**

예외상황을 파악하고 문제를 해결해서 정상상태로 돌려놓는방법

예외를 잡아서 일정시간, 조건만큼 대기하고 다시 재시도 반복

최대 재시도 횟수를 넘기게 될 경우 예외 발생

```
final int MAX_RETRY = 100;
public Object someMethod() {
    int maxRetry = MAX_RETRY;
    while(maxRetry > 0) {
        try {
            ...
        } catch(SomeException e) {
            // 로그 출력. 정해진 시간만큼 대기한다.
        } finally {
            // 리소스 반납 및 정리 작업
        }
    }
    // 최대 재시도 횟수를 넘기면 직접 예외를 발생시킨다.
    throw new RetryFailedException();
}
```

**2) 회피**

예외처리를 직접 담당하지 않고 호출한 쪽으로 던져 회피

**3) 전환**

예외 회피와 비슷하게 메서드 밖으로 예외를 던지지만, 그냥 던지지 않고 적절한 예외로 전환해서 넘기는 방법

# 📌NIO

- 읽는 데이터를 무조건 buffer에 저장해서 버퍼내 데이터의 위치를 이동해 필요한 부분만 읽고 쓴다.
- 데이터 IO를 channel을 통해 읽는다.
  - channel : non-blocking read를 할 수 있도록 지원하는 connection
  - 양방향으로 입력과 출력이 가능하다.
  - 채널에서 데이터를 주고 받을 때 사용하는 것이 버퍼이다.
  - ServerSocketChannel과 SocketChannel을 이용하여 TCP 네트워크 프로그램에 이용한다.
- 넌블로킹, 블로킹 둘다 가짐
  - 파일을 읽는 File I/O 는 blocking. 그 외는 non blocking
- selector
  - 여러개의 **채널에서 발생하는 이벤트**(연결이 생성됨, 데이터가 도착함 등)를 **모니터링할 수 있는 객체**다. 하나의 selector(스레드)에서 여러 채널에 대해 지속적으로 모니터링 한다.
  - 실제 수신되는 데이터가 없음에도 무한루프를 돌며 버퍼에 데이터 여부를 계속 확인하게 되는데 이를 방지하기 위해 사용된다.

- 연결 클라이언트가 많고 IO가 작은 경우
  - NIO는 버퍼할당크기가 문제가 되고, 모든 입출력 작업에 버퍼를 무조건 사용해야 하므로 즉시 처리하는 IO보다 성능 저하가 발생할 수 있다.

# 📌cms 순서

1) **initial mark** : GC Root로부터 바로 참조되거나, young 영역의 살아있는 객체로부터 바로 참조되는 old 영역의 객체를 mark (STW 발생)

2) **Concurrent mark** : initial mark단계에서 mark한 객체부터 시작해서 **old영역을 순회**하면서 살아있는 모든 객체들을 markt한다. ( STW 발생하지 않고 애플리케이션 스레드를 멈추지 않고 동작)

3) **Concurrent reclean** :

**Card라는 힙영역 안에 dirty로 표시한 객체로부터 참조되고 있는 객체를 mark** ( STW 발생하지 않고 애플리케이션 스레드를 멈추지 않고 동작)

4) **Concurrent aborable Preclean**

변화한 객체들을 계속 **스캔**

( STW 발생하지 않고 애플리케이션 스레드를 멈추지 않고 동작, Final Remark의 수행시간을 최대한 짧게 만든다)

5) **Final Remark**

STW를 통해 애플리케이션 스레드를 잠시 멈춤으로써 **객체들의 상태를 완전히 반영**한다. 이 단계를 young 영역이 거의 비워져있을 때 수행하게 하도록 하여 STW을 일으키는 동작이 연쇄적으로 발생하는 것을 방지

6) **Concurrent sweep**

애플리케이션 스레드와 병렬적으로 수행, 사용하지 않는 객체들을 정리하여 **빈공간을 확보**한다

→ CMS는 컴팩션 안한다