

YC Tech

웹 백엔드 실무 개발 프로젝트




연세대학교 미래교육원
YC (Yonsei X Codepresso) Tech Academy

2주차 recap



Unit test

 Mockito 를 이용한 단위 테스트 코드 작성

```
private PostService postService;
private PostRepository postRepository = mock(PostRepository.class);

Post post1 = new Post(LocalDate.now().minusDays(2));
Post post2 = new Post(LocalDate.now().plusDays(2));

List<Post> stubPosts = List.of(post1, post2);

// stubbing : mock 객체를 통해 원하는 값 return
when(postRepository.findAll()).thenReturn(stubPosts);
// 테스트 대상의 return 값
List<Post> posts = postService.getPostList();
// 예상대로 정렬이 되었는지 검증
assertTrue(isSortedDescending(posts));
```

Post 조회 / 전체 조회 / 삭제 구현 & 테스트

1. Post 단 건 조회 기능 (GET /posts/{postId})
 - a. service 구현
 - b. 테스트 코드 작성
 - c. `postRepository.findById` 가 호출되는지 검증
2. Post 전체 조회 기능 (GET /posts)
 - a. controller 와 service 연동
3. Post 삭제 기능(?? ??)
 - a. controller 스펙 정의
 - b. service 구현
 - c. `postRepository.??` 호출되는지 검증

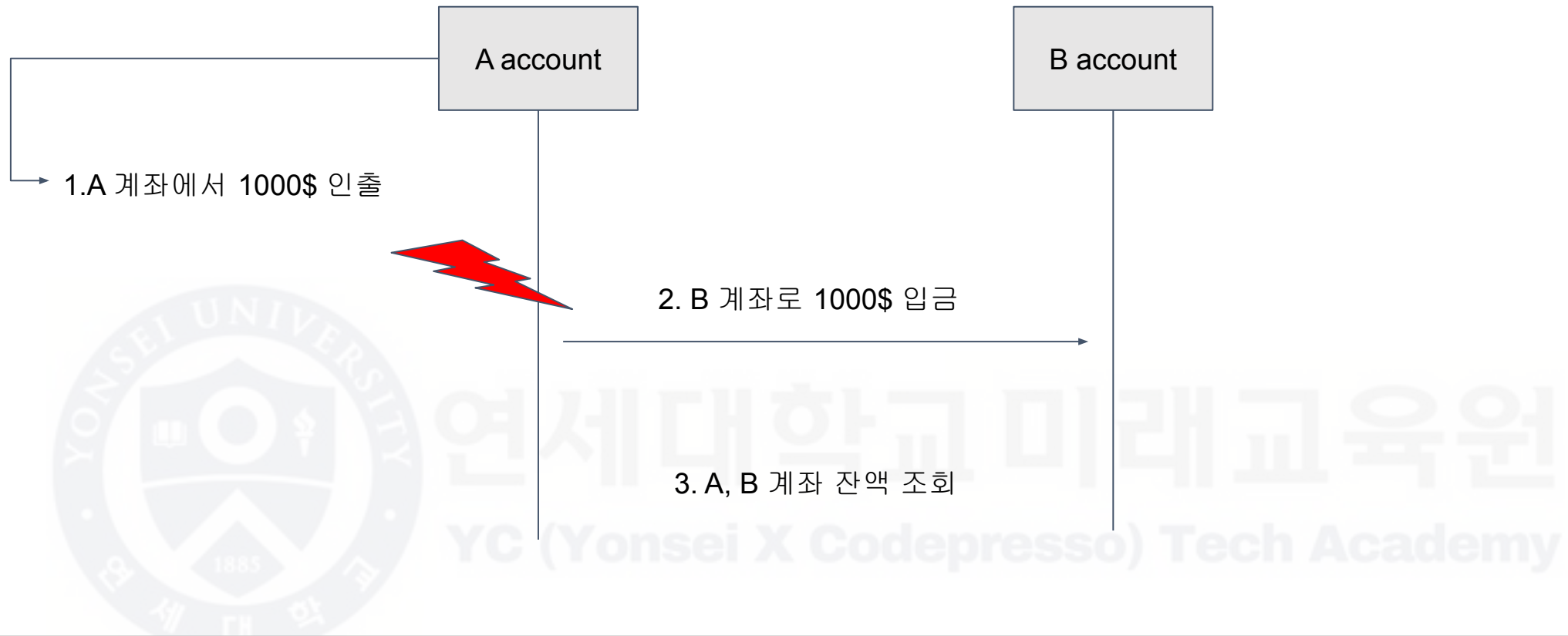
RDBMS



특징

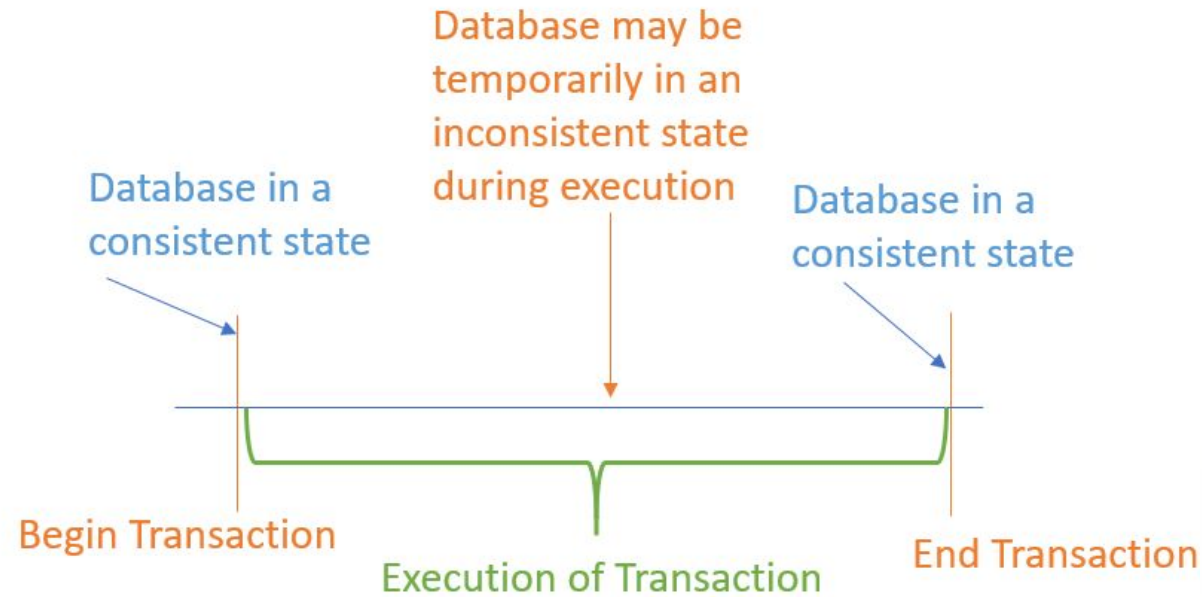
- 구조화된 쿼리 언어 (SQL): RDBMS의 가장 강력한 이점은 데이터를 저장, 검색, 조작 및 관리하기 위해 SQL을 사용할 수 있음.
- 원자성: RDBMS의 이 기능은 시스템의 모든 트랜잭션이 원자적으로 완료되도록 보장. 모든 트랜잭션 작업이 성공적으로 수행되거나 하나도 수행되지 않는 것을 의미하며 데이터 무결성과 일관성을 보장
- 신뢰성: RDBMS는 데이터를 저장, 업데이트 및 검색하기 위한 신뢰할 수 있는 시스템을 제공
- 확장성: RDBMS는 높은 확장성을 제공하며 큰 데이터 집합을 수용할 수 있도록 쉽게 확장할 수 있음
- 유연성: RDBMS는 데이터를 쉽게 추가, 삭제 및 업데이트할 수 있는 높은 유연성을 제공

ACID



- **Atomicity** : 원자성은 트랜잭션과 관련된 작업들이 부분적으로 실행되다가 중지되는 것이 아니라 하나의 원자 단위로 수행되는 것을 보장하는 특징입니다. 즉, 중간 단계까지 실행되는 것이 아니라 처음부터 끝까지 완전하게 실행되며 중간에서 실패하는 일이 없도록 합니다.
- 중간에 송금이 실패하는 경우 1 부터 다시 시작, 1~3까지의 과정이 하나의 단위로 수행되도록 보장

ACID



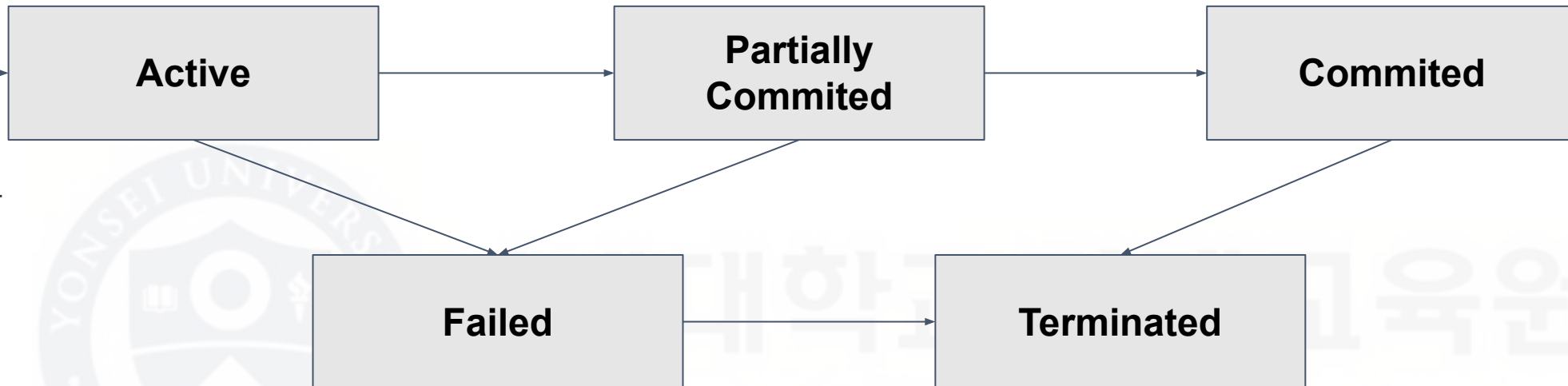
- **Consistency** : 일관성은 트랜잭션이 완료되면 언제나 일관된 DB 상태를 유지하는 것을 의미합니다.
- **Isolation** : 독립성은 트랜잭션을 수행 할 시, 다른 트랜잭션의 연산 작업이 끼어들지 못하도록 보장하는 것을 의미하는데, 다시말해 다른 트랜잭션의 연산이 중간 단계의 데이터를 볼 수 없음을 의미합니다.
- **Durability** : 지속성은 성공적인 트랜잭션은 영원히 반영되어야 함을 의미합니다. 시스템 에러, DB 일관성 체크 등을 하더라도 유지되어야함을 의미합니다. 트랜잭션은 로그에 모든 것이 저장된 후에만 **Commit** 상태로 간주될 수 있습니다.

Transaction

1

read, write

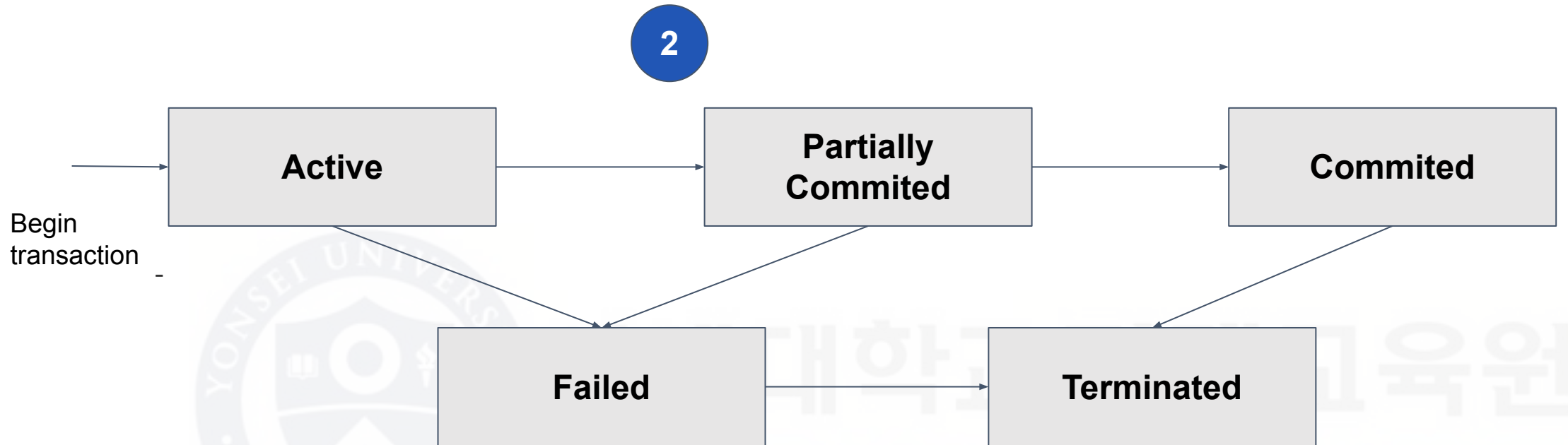
Begin transaction -



① Active

- read,write 를 수행하는 상태, 작업이 완료될때까지 유지된다.

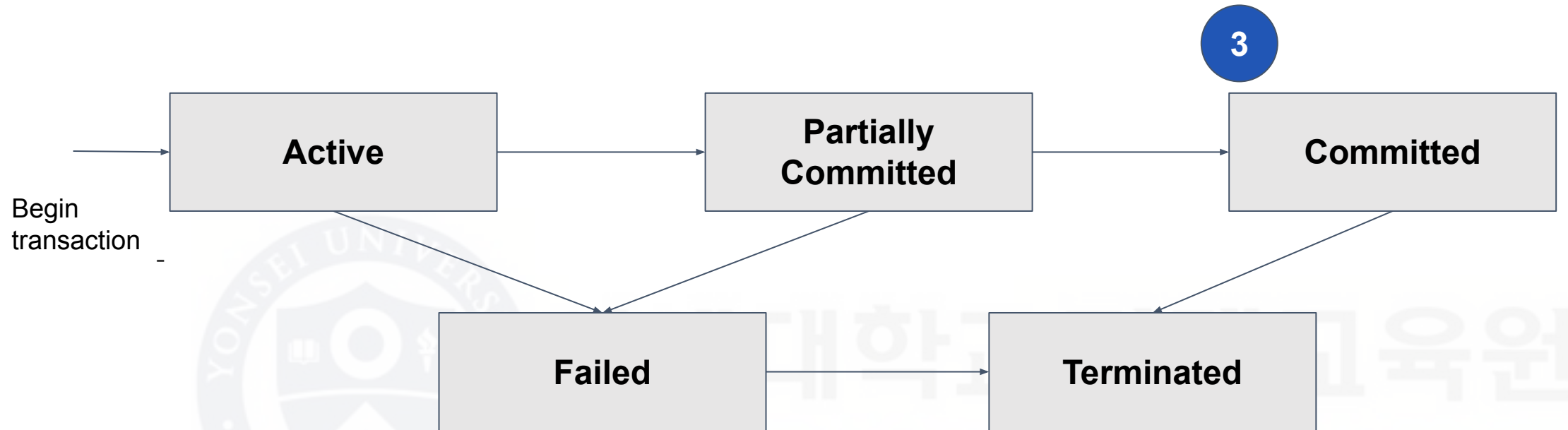
Transaction



② Partially Committed

- 변경이 메모리 버퍼 내에서 발생한다. 아직 변경된 부분을 디스크에 쓰지 않은 상태

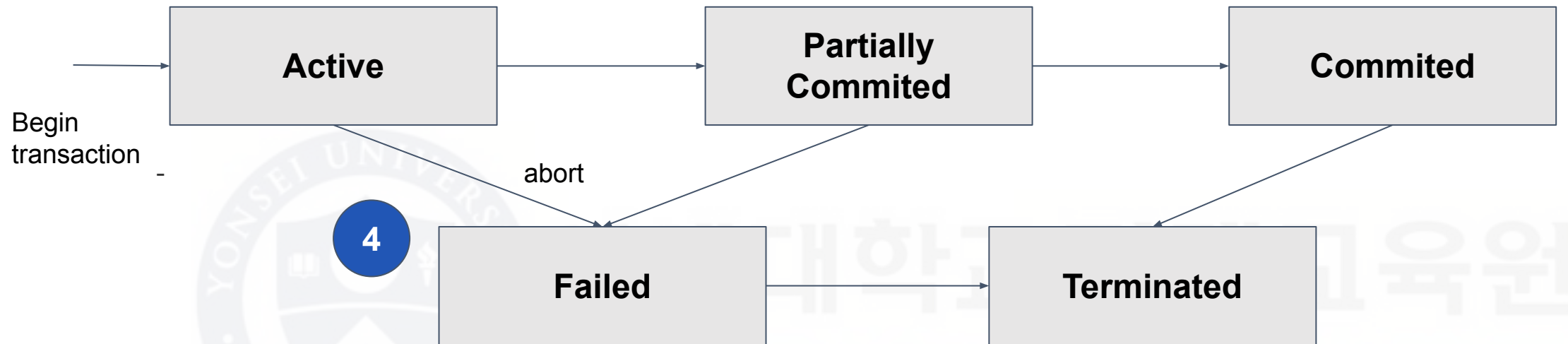
Transaction



③ Committed

- 디스크에 영구적으로 저장된 상태. `rollback` 이 불가능한 상태

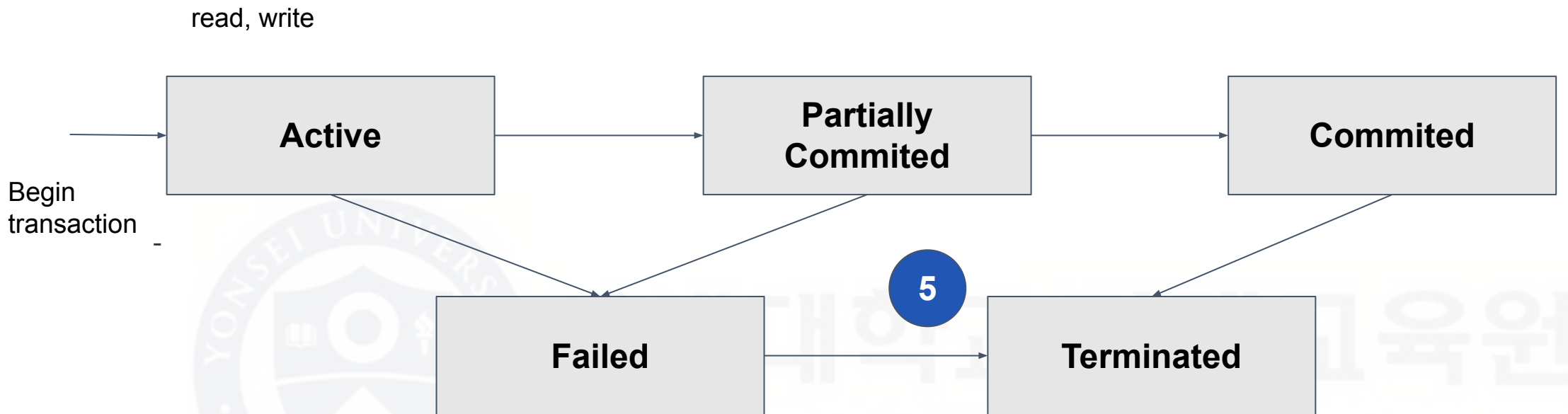
Transaction



④ Failed

- 디스크 이슈, 메모리 이슈 등 어떠한 이유로 실패를 한 경우, **rollback** 을 할지 그대로 **transaction** 을 종료시킬 수도 있다.

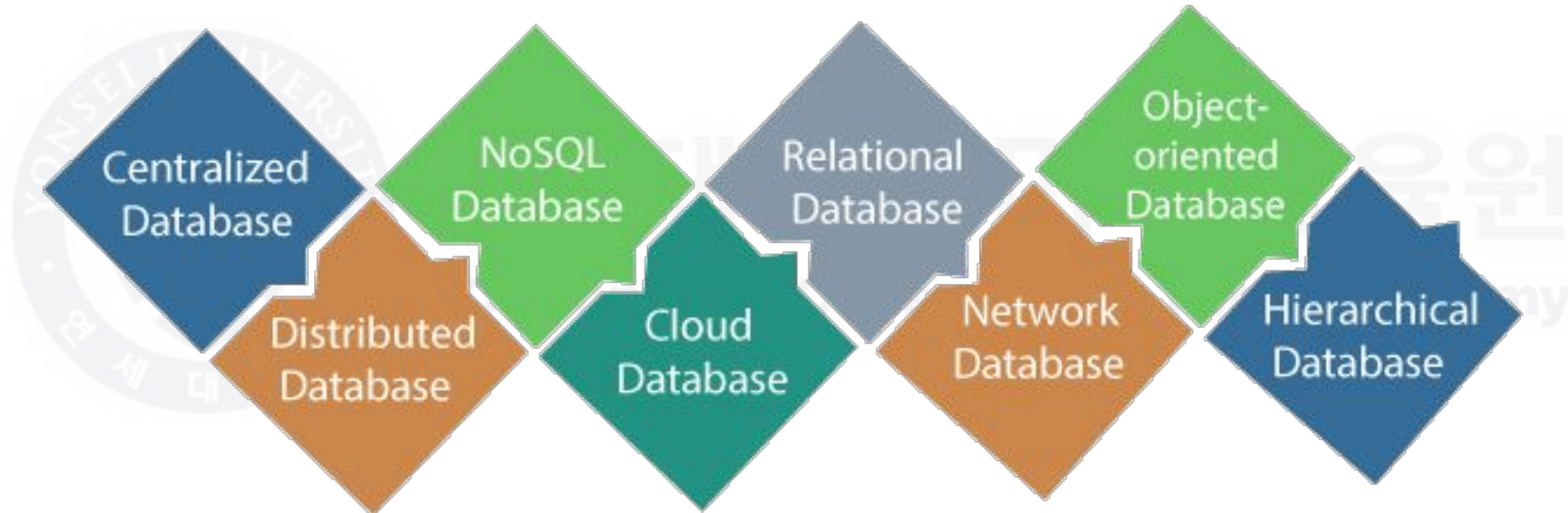
Transaction



⑤ Terminated

- 성공 혹은 실패 후 종료 상태

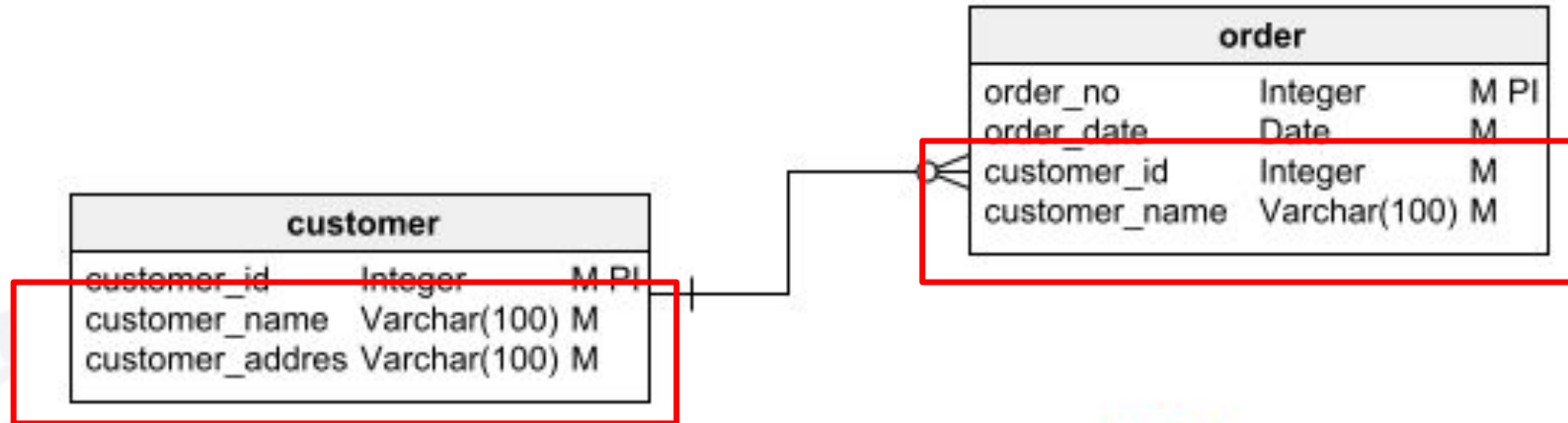
Types of Database



Database design



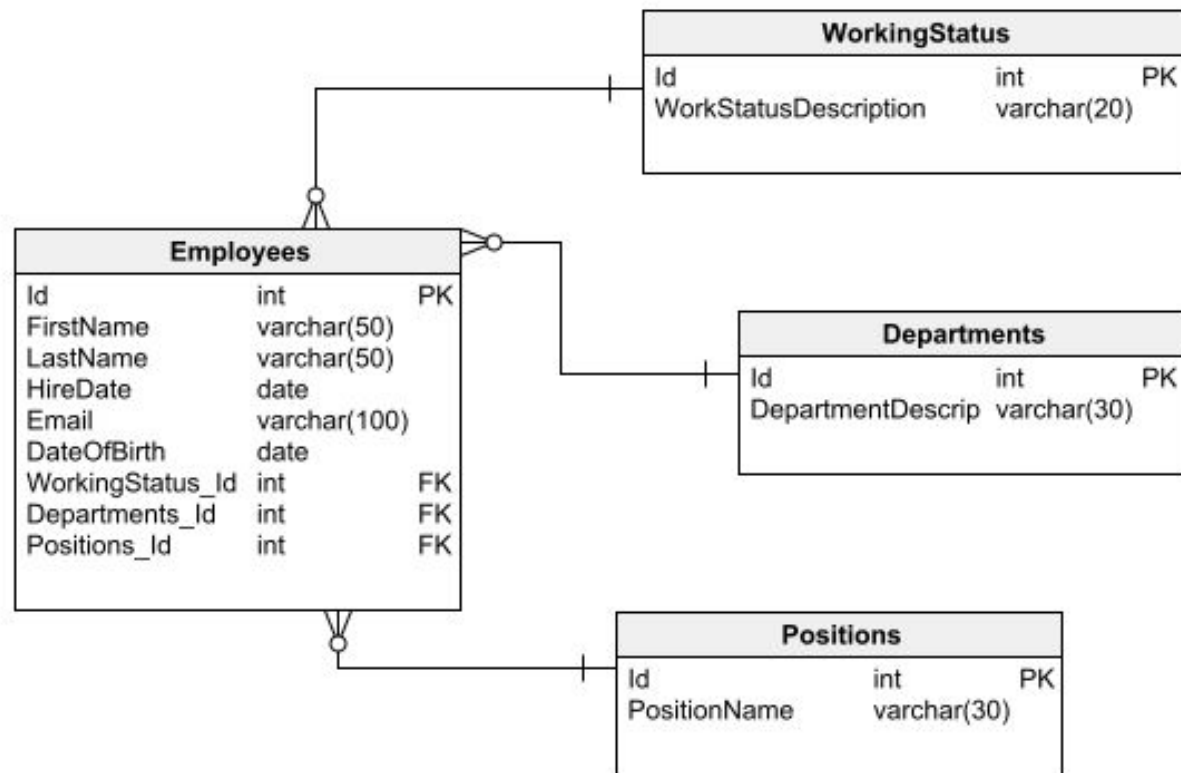
Database design



중복 최소화

- 같은 데이터가 여러 테이블에 걸쳐 저장되었으면 **inconsistency** 문제가 생길 수 있음
 - 만약 어느 한 쪽 테이블의 데이터는 업데이트 되었는데 다른 테이블은 업데이트가 안되는 경우
 - 저장 공간을 불필요하게 증가 시킴
- 예외 사항
 - 분석을 위한 테이블의 중복
 - 변경이 일어날 가능성이 적은 데이터

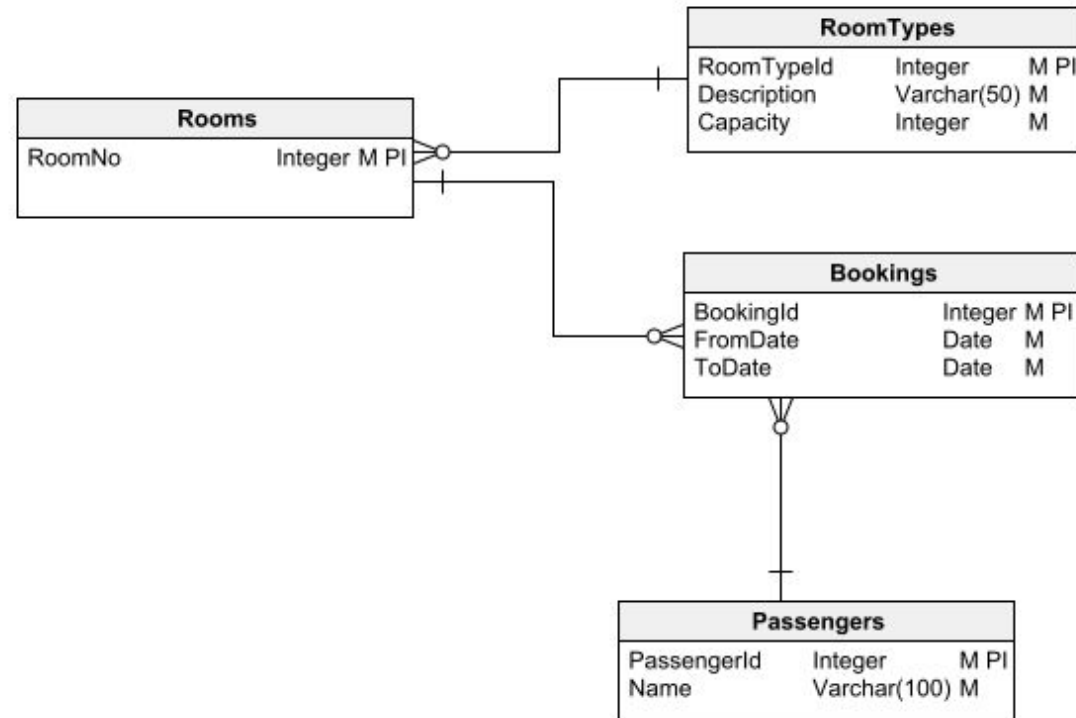
Database design



 primary key

- 데이터의 **unique** 함을 나타내기 위한 키
- 관계 표현을 위해 필수적

Database design



Referential Integrity

- primary key와 foreign key 를 활용하여 데이터 관계 무결성을 보장
- table 검색 시 속도 향상

Database design

Customer #	Customer name	Customer Address
1001	Kelly Hodge	5331 Rexford Court, Montgomery AL 36116
4195	Jameson Doe	5331 Rexford Court, Montgomery AL 36116
3412	Brayden Huang	5331 Rexford Court, Montgomery AL 36116




Atomicity

- 유지 보수 성을 위해 작은 부분으로 나누어서 저장하는 것이 바람직함
 - name -> firstname, lastname, address -> city, state, postal code
- 나중에 관리를 위해 분리하는 것은 어려움

Database design

Customer #	Customer name	Customer Birthdate
1001	Kelly Hodge	May 14, 1977
4195	Jameson Doe	2001-06-18
3412	Brayden Huang	12/20/1987

 적절한 type 사용

- string 으로 저장 후 parsing 하는 경우 정규화와 마이그레이션이 어려울 수 있음
- index 의 성능이 떨어짐

Database design


특징

- **@Controller** : 스프링 MVC 컨트롤러로 인식된다.
- **@Repository** : 스프링 데이터 접근 계층으로 인식하고 해당 계층에서 발생하는 예외는 모두 **DataAccessException**으로 변환한다.
- **@Service** : 특별한 처리는 하지 않으나, 개발자들이 핵심 비즈니스 계층을 인식하는데 도움을 준다.
- **@Configuration** : 스프링 설정 정보로 인식하고 **bean** 등록, **singleton scope** 을 보장

use case

- **@Bean** 개발자가 컨트롤이 불가능한 외부 라이브러리들을 **Bean**으로 등록하고 싶은 경우 에 사용된다.
 - 메소드 또는 어노테이션 단위에 붙일 수 있다.
- **@Component** 개발자가 직접 컨트롤이 가능한 클래스들의 경우에 사용된다.
 - 클래스 또는 인터페이스 단위에 붙일 수 있다.

Database design

Customer #	Customer name	Customer Personal Information
1001	Kelly Hodge	
4195	Jameson Doe	
3412	Brayden Huang	



민감 정보는 적절한 보안 처리

- 비밀번호, 개인정보 등은 **hasing** 처리
- 권한 별로 접근 가능한 **schema** 분리

Database design practice

- 책: 각 책은 제목, **ISBN** (국제 표준 도서 번호), 가격, 출판일 및 형식 (실물, **eBook**, 오디오북)을 가지고 있습니다. 각 책은 하나 이상의 저자와 연관되어 있습니다. 책은 또한 하나 이상의 장르 (예: 소설, 비소설, 공상 과학)에 속할 수 있습니다.
- 저자: 저자는 이름과 고유한 저자 **ID**를 가지고 있습니다. 저자는 여러 책을 쓸 수 있습니다.
- 출판사: 출판사는 이름과 고유한 출판사 **ID**를 가지고 있습니다. 출판사는 여러 책을 출판할 수 있습니다.
- 고객: 고객은 이름, 이메일 주소, 배송 주소 및 고유한 고객 **ID**를 가지고 있습니다. 고객은 계정을 생성하고 로그인하며 구매를 할 수 있습니다.
- 장바구니: 각 고객은 상품을 추가하고 제거할 수 있는 장바구니를 가질 수 있습니다. 장바구니는 상품에 대한 정보 (예: 책 **ID**, 수량, 형식)를 저장해야 합니다.
- 주문: 고객이 구매를 할 때 주문이 생성됩니다. 주문에는 주문한 고객에 대한 정보, 주문 일시 및 총 비용이 포함되어 있습니다. 각 주문에는 여러 상품이 포함될 수 있습니다.
- 리뷰: 고객은 책에 대한 리뷰를 남길 수 있습니다. 각 리뷰에는 등급 (예: 1에서 5까지의 별점)과 선택적인 텍스트 내용이 있을 수 있습니다.

SQL





DDL(Data Definition Language, 데이터 정의 언어)

- **DDL**은 데이터베이스 스키마와 설명을 처리하도록 정의하는 언어.
- 데이터베이스나 테이블 생성/변경/삭제 등의 작업이 포함

CREATE

- 데이터베이스 개체(테이블, 인덱스, 제약조건 등)의 정의

DROP

- 데이터베이스 개체 삭제

ALTER

- 데이터베이스 개체 정의 변경



DML(Data Manipulation Language, 데이터 조작 언어)

- 데이터 검색, 삽입, 변경, 삭제 수행을 조작하는 언어
- 실질적으로 저장된 데이터를 관리하고 처리할 때 사용

SELECT

- 테이블 데이터의 검색 결과 집합의 취득

INSERT

- 행 데이터 또는 테이블 데이터의 삽입

DELETE

- 데이터 삭제

UPDATE

- 데이터 업데이트



DCL(Data Control Language, 데이터 제어 언어)

- 데이터 검색, 삽입, 변경, 삭제 수행을 조작하는 언어
- 실질적으로 저장된 데이터를 관리하고 처리할 때 사용

COMMIT

- transaction 결과 반영

ROLLBACK

- transaction 작업 취소

GRANT

- 사용자 권한 부여

REVOKE

- 사용자 권한 취소

SQL Practice

CountryCode	Country
AU	Australia
CN	China
DE	Germany
FR	France
HK	Hong Kong
JP	Japan
SG	Singapore
UK	United Kingdom
US	United States of America

CountryCode	Country
UK	United Kingdom
US	United States of America

CountryCode 가 'U' 로 시작하는 결과가 나오게 쿼리를 작성해보시오

PONumber	POItem	DateOfDelivery	QuantityDelivered
C0001	01	2018-03-12	300
C0001	02	2018-03-13	600
C0001	02	2018-04-17	1300
C0002	01	2018-04-24	250
C0003	01	2018-05-03	1400
C0003	02	2018-05-10	1200

ponumber	poitem	Qty Delivered
C0001	01	300
C0001	02	1900
C0002	01	250
C0003	01	1400
C0003	02	1200

ponumber/poitem 의 합계가 나오도록 조회하는 쿼리를 작성하시오

game

id	mdate	stadium	team1	team2
1001	8 June 2012	National Stadium, Warsaw	POL	GRE
1002	8 June 2012	Stadion Miejski (Wroclaw)	RUS	CZE
1003	12 June 2012	Stadion Miejski (Wroclaw)	GRE	CZE
1004	12 June 2012	National Stadium, Warsaw	POL	RUS

goal

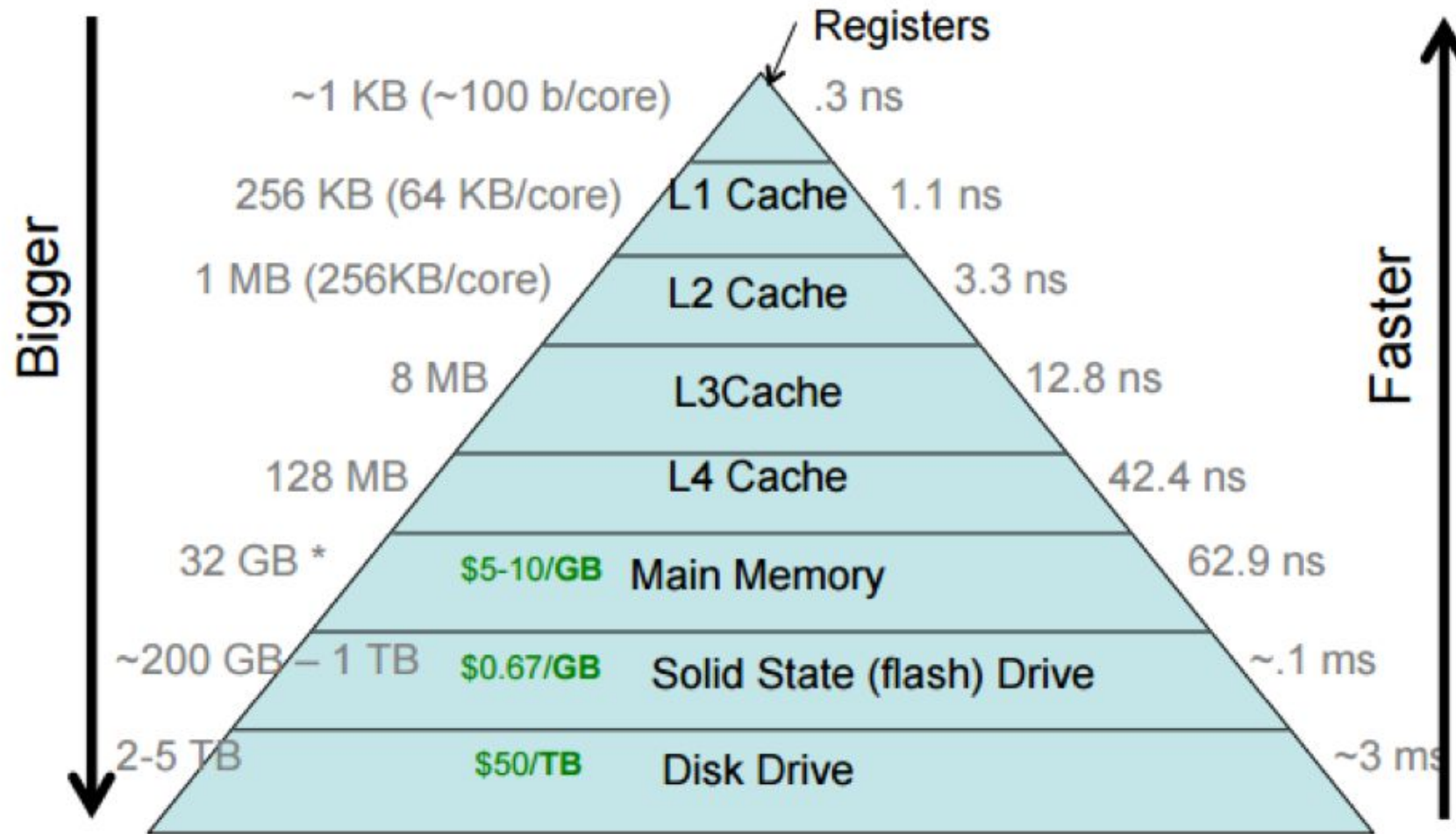
matchid	teamid	player	gtime
1001	POL	Robert Lewandowski	17
1001	GRE	Dimitris Salpingidis	51
1002	RUS	Alan Dzagoev	15
1001	RUS	Roman Pavlyuchenko	82
...			

1. goal 테이블의 선언하는 DDL 쿼리를 수행하시오.
2. GRE (greece) 상대로 득점을 한 선수와 그 골의 갯수를 구하는 쿼리를 작성하시오.

Index



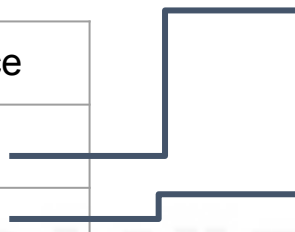
Performance vs Space



<https://cs61.seas.harvard.edu/site/img/storage-hierarchy.png>

Index

Index key	Data Reference			
1		1	A	Alpha
		2	B	Bravo
3		3	C	Charlie
		4	D	Delta



The diagram illustrates a B-tree index structure. On the left, an index table has two rows: (1, Data Reference) and (3, Data Reference). On the right, a data table has four rows: (1, A, Alpha), (2, B, Bravo), (3, C, Charlie), and (4, D, Delta). A line connects the index key '1' to the first data row (1, A, Alpha). Another line connects the index key '3' to the third data row (3, C, Charlie). The lines are drawn as horizontal segments from the index table, followed by vertical segments pointing to the corresponding data rows.

- 조회 성능을 향상 시키기 위한 자료구조
- **key**는 테이블의 설정된 **column** 값을 가짐. **key** 를 비교하여 실제 **database** 의 데이터 검색
- **B+ tree** 자료구조 이용
- 데이터가 지속적으로 업데이트 되는 경우 **B+ tree** 을 균등하게 유지하는 것이 중요함
- **B++ tree** 자료구조 계산 후 **create/update** 하므로 쓰기 성능은 저하

Index



post 전체 목록 조회

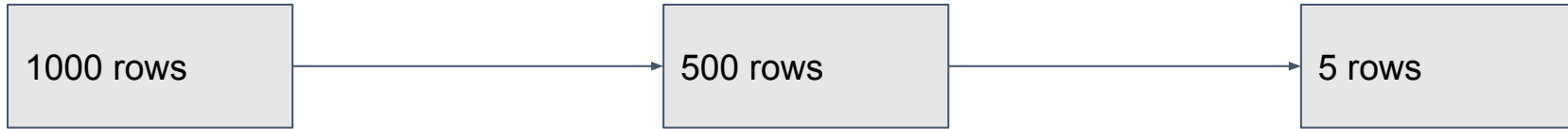
- 생성 날짜 내림차순으로 조회 (updated_at DESC)

id	title	content	updated_at
1	~	~	2010-01-23
2	~	~	2022-02-11
...			
1000	~		2016-03-23

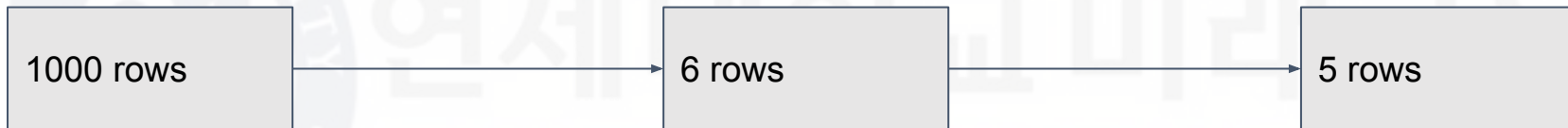
Composite index

‘Blake’ 라는 ‘남성’ 을 검색

(gender, name)



(name, gender)



 composite index 의 순서가 중요

- 최대한 검색에서 많은 데이터를 필터링 하도록 하는 것이 성능에 유리
- 위의 예시에서는 이름, 성별 순서로 **composite index** 를 생성하는 것이 적합한 설계

Index design

index 생성시 고려해야할 사항

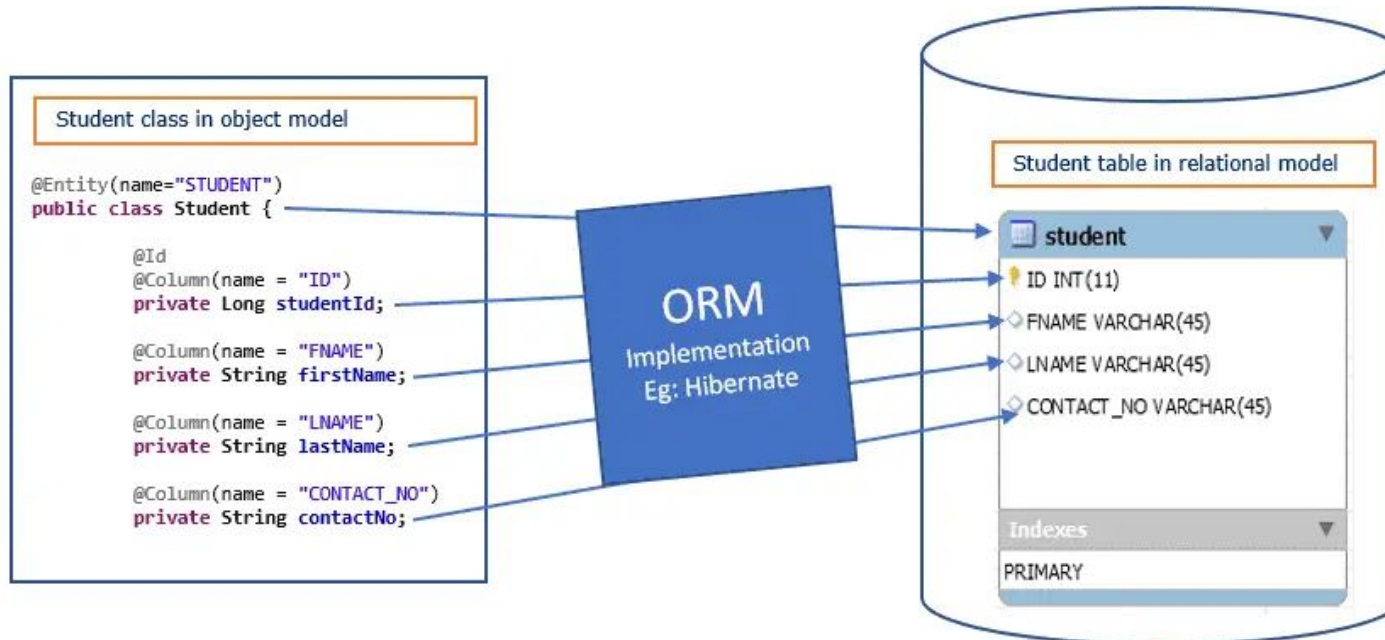
- **key** 의 크기를 작게할 수록 성능 향상
- 단일 **index** 여러 개 보다 **multi index** 생성 고려
- **join** 시 사용되는 **column** 은 **index**를 구성하는 것을 고려
- **update**가 빈번하지 않은 **column** 은 **index**를 구성하는 것을 고려
- 조건 절(**where**) 에 자주 사용되는 **column** 은 **index**를 구성하는 것을 고려
- **selectivity** ($\# \text{ of distinct values} / n$) 작을 경우 **index scan** 성능이 우수
 - o 텍스트 같이 **non-deterministic** 한 **column** 은 DB 에서 제공하는 다른 유형의 **index** 사용



ORM



Object Relational Mapping



- **Object-Relational Mapping** (객체와 관계형데이터베이스 매핑, 객체와 DB의 테이블이 매핑을 이루는 것) - 객체가 테이블이 되도록 매핑 시켜주는 프레임워크 이다.
- 프로그램의 복잡도를 줄이고 자바 객체와 쿼리를 분리할 수 있으며 트랜잭션 처리나 기타 데이터베이스 관련 작업들을 좀 더 편리하게 처리할 수 있는 방법
- **SQL Query**가 아닌 직관적인 코드(메서드)로서 데이터를 조작할 수 있다.
ex) 기존쿼리 : **SELECT * FROM MEMBER;**
- 이를 **ORM**을 사용하면 **Member**테이블과 매핑된 객체가 **member**라고 할 때, **member.findAll()**이라는 메서드 호출로 데이터 조회가 가능하다.

Object Relational Mapping

ORM 사용 시 장점

- 객체 지향적인 코드로 인해 더 직관적이고 비즈니스 로직에 더 집중할 수 있게 도와준다.
- 선언문, 할당, 종료 같은 부수적인 코드가 없거나 급격히 줄어든다.
- 각종 객체에 대한 코드를 별도로 작성하기 때문에 코드의 가독성을 올려준다. **SQL**의 절차적이고 순차적인 접근이 아닌 객체 지향적인 접근으로 인해 생산성이 증가
- 매핑정보가 명확하여, **ERD**를 보는 것에 대한 의존도를 낮출 수 있다. **DBMS**에 대한 종속성이 줄어든다. 대부분 **ORM** 솔루션은 **DB**에 종속적이지 않다.

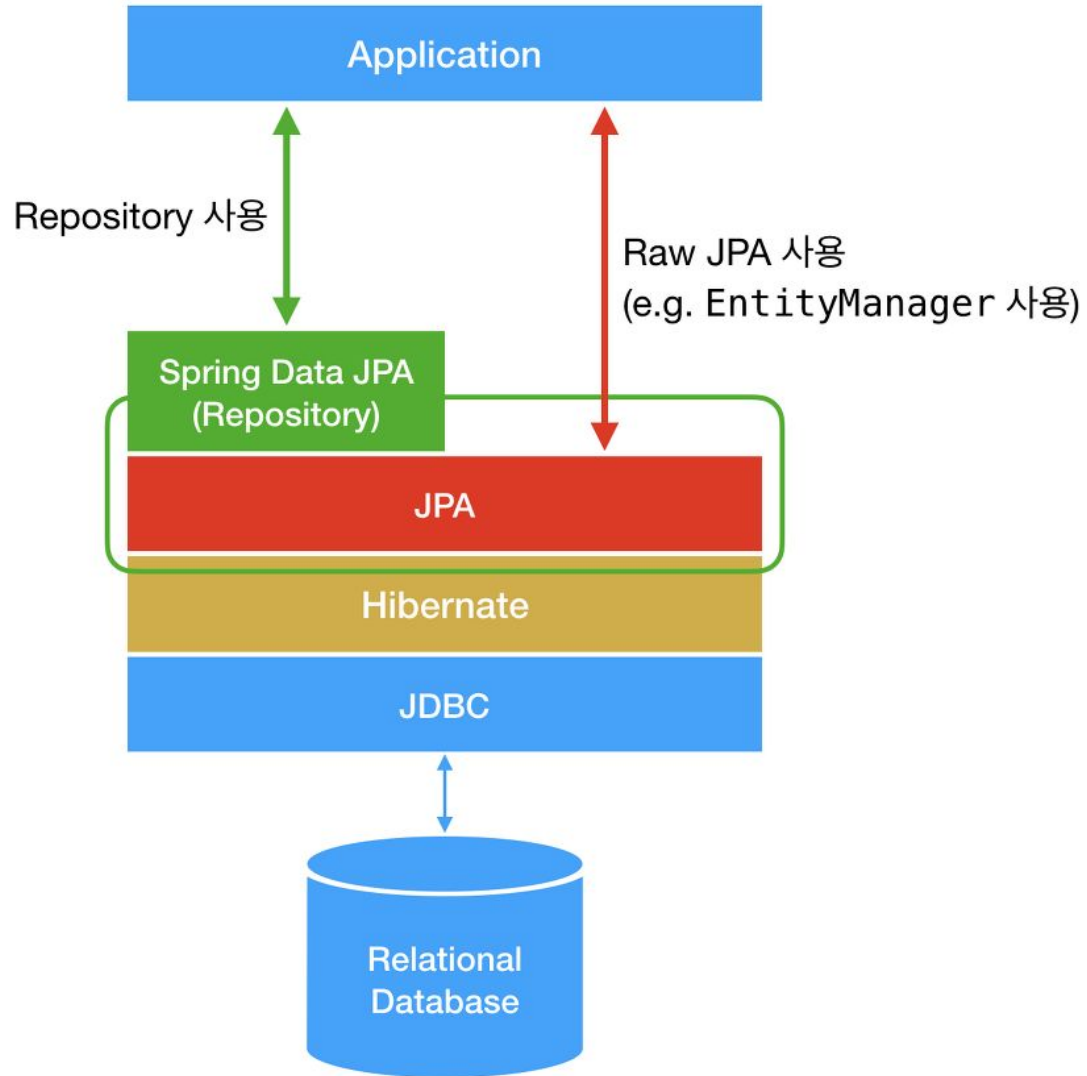
ORM 사용 시 단점

- 프로젝트의 복잡성이 커질 경우 난이도 또한 올라갈 수 있다.
- 잘못 구현된 경우에 속도 저하 및 심각할 경우 일관성이 보장이 안되는 문제점이 생길 수 있다.

JPA



JPA



- JPA란 자바 ORM 기술에 대한 API 표준 명세를 의미합니다. JPA는 ORM을 사용하기 위한 인터페이스를 모아둔 것이며, JPA를 사용하기 위해서는 JPA를 구현한 **Hibernate, EclipseLink, DataNucleus** 같은 ORM 프레임워크를 사용해야 합니다.
- **Hibernate** : JPA를 구현한 구현체
- **JDBC** : 자바 프로그래밍 언어와 다양한 데이터베이스 SQL 또는 테이블 형태의 데이터 사이에 독립적인 연결을 지원하는 표준

jpa:

hibernate:

ddl-auto: **create** #create-drop, update, validate, none




ddl-auto 옵션

- spring boot 초기 구동 시 데이터베이스 초기화 전략 설정
 - **none** : 사용하지 않음
 - **create** : 기존 테이블 삭제 후 테이블 생성
 - **create-drop** : 기존 테이블 삭제 후 테이블 생성, 종료 시점에 테이블 삭제
 - **update** : 변경된 스키마 적용
 - **validate** : 엔티티와 테이블 정상 매핑 확인
- **validate** 외에는 **live** 에서 사용하면 장애 발생시킬 수 있음

@Repository method

```
public interface UserRepository extends Repository<User, Long> {  
  
    // where u.emailAddress = ?1 and u.lastname = ?2 와 같은 쿼리 구문  
    List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);  
  
}
```

 method naming 으로 query 작성

- method naming 에 parameter 를 포함하여 query 작성하는 기능 지원
ref) <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods>

Native query

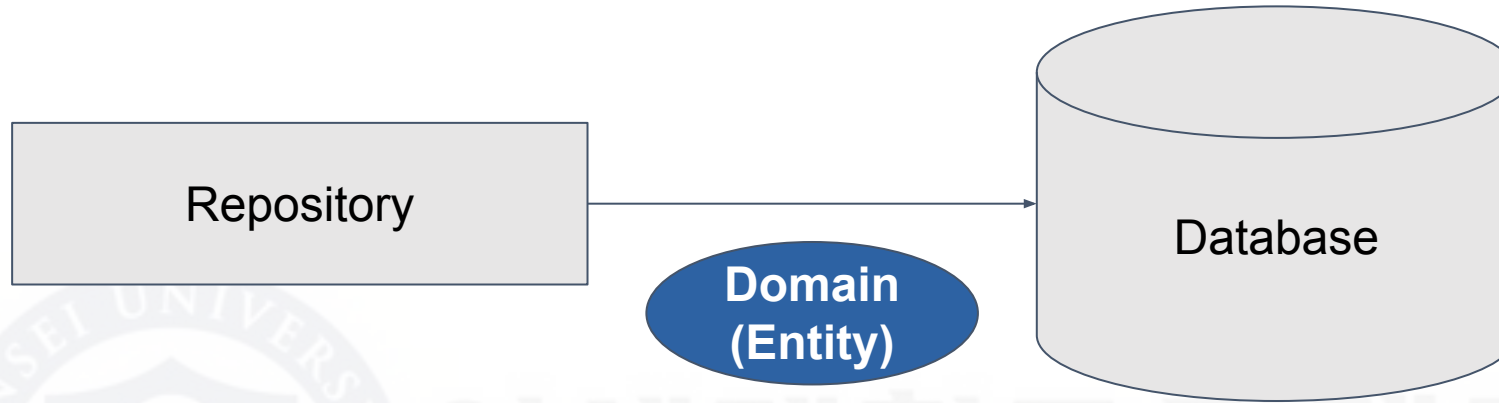
```
public interface SnackRepository extends JpaRepository<Snack, Integer>{  
  
    // 일반 SQL 쿼리  
    @Query(value = "select snack_id, name, price from snack", nativeQuery = true)  
    public List<Snack> selectAllSQL();  
}
```

- method의 parameter를 통해 의존성을 주입하는 방식
- Setter Injection은 선택적인 의존성을 사용할 때 유용하다. 상황에 따라 의존성 주입이 가능

```
public interface SnackRepository extends JpaRepository<Snack, Integer>{ // 제네릭 타입: <엔티티 클래스, 엔티티클래스의  
기본키>  
    // 일반 JPQL쿼리, from뒤는 엔티티 명 (소문자로 할 시 에러)  
    @Query(value = "select sn from Snack sn")  
    public List<Snack> selectAllJPQL();  
}
```

- 일반 JPQL 을 지원하므로 객체명으로 테이블을 지정해야함

Entity



- 실제 DB의 테이블과 매칭될 클래스이다. 즉, 가장 DB의 테이블과 가깝다고 할 수 있다. `@Entity`, `@Id`, `@Column`, `@GeneratedValue` 등의 애노테이션을 사용한다.
- 외부에서 **Setter**를 이용한 값의 변경으로 DB에 오류가 생길 수 있다. 그렇기에 **Setter**는 닫아두고 **Getter**만 열어두어야 한다.
- **View** 계층과 **DB** 계층의 분리가 확실해야 한다.

Entity 간 Relation 표현

JPA 는 연관관계에 있는 상대 테이블의 **primary key** 를 **member variable** 로 갖지 않고, **entity** 를 객체 자체를 참조

```
// Mybatis  
private Integer categoryNo;  
  
// JPA  
private Category category
```

- Many To One - 다대일 (N : 1)
- One To Many - 일대다 (1 : N)
- One To One - 일대일 (1 : 1)
- Many To Many - 다대다 (N : N)

Entity 간 Relation 표현

관계를 아래 유형 중 하나로 표현 가능

- **Many To One** - 다대일($N : 1$)
 - ex) 책 : 카테고리
- **One To Many** - 일대다($1 : N$)
 - ex) 카테고리 : 책
- **One To One** - 일대일($1 : 1$)
 - ex) 책 : ISBN
- **Many To Many** - 다대다($N : N$)
 - ex) 책 : 작가

Entity 간 Relation 표현 - 단방향

```
@Entity
@Table(name="category")
public class Category {
    @Id
    @Column(name="no")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer no;
}

@Entity
@Table( name="book")
public class Book {
    @Id
    @Column(name="no")
    @GeneratedValue( strategy = GenerationType.IDENTITY )
    private Integer no;

    @ManyToOne
    @JoinColumn(name ="category_no")
    private Category category;
}
```



어떤 Entity 를 중심으로 상대 Entity 의 관점으로 참조하느냐에 따라 단방향/양방향이 결정

- Book entity 는 Category 에 대한 참조를 할 수 있지만 Category 는 참조할 수 없음
- 한 쪽의 entity 가 상대 entity 를 참조
- 단방향

Entity 간 Relation 표현 - 양방향

```
@Entity
@Table(name="category")
public class Category {
    @Id
    @Column(name="no")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer no;

    @Column( name="name", nullable=false, length=100 )
    private String name;

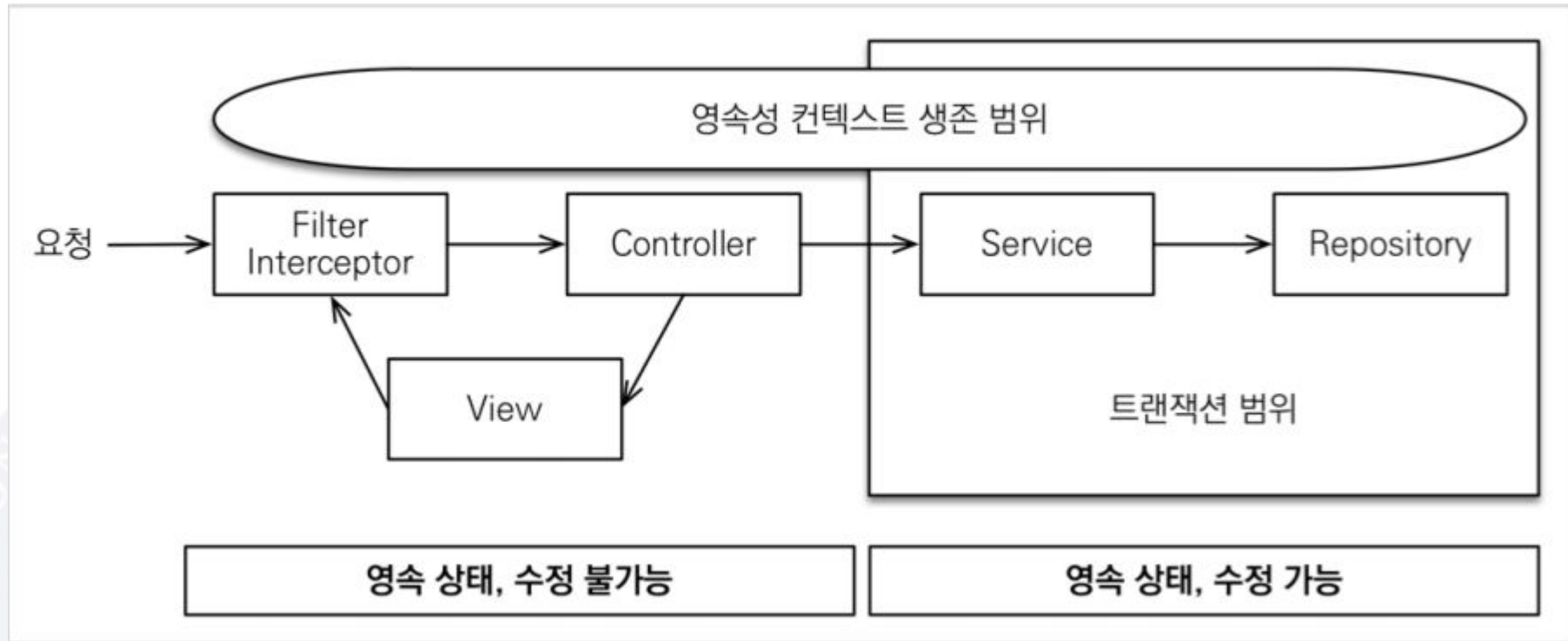
    @OneToMany(mappedBy="category")
    private List<Book> books = new ArrayList<Book>();
}
```



어떤 Entity 를 중심으로 상대 Entity 의 관점으로 참조하느냐에 따라 단방향/양방향이 결정

- Category 는 Book entity 를 List 로 참조
- 주인 entity 가 아닌 경우 mappedBy 속성을 이용해 속성의 값으로 연관관계의 주인을 설정
- 양방향

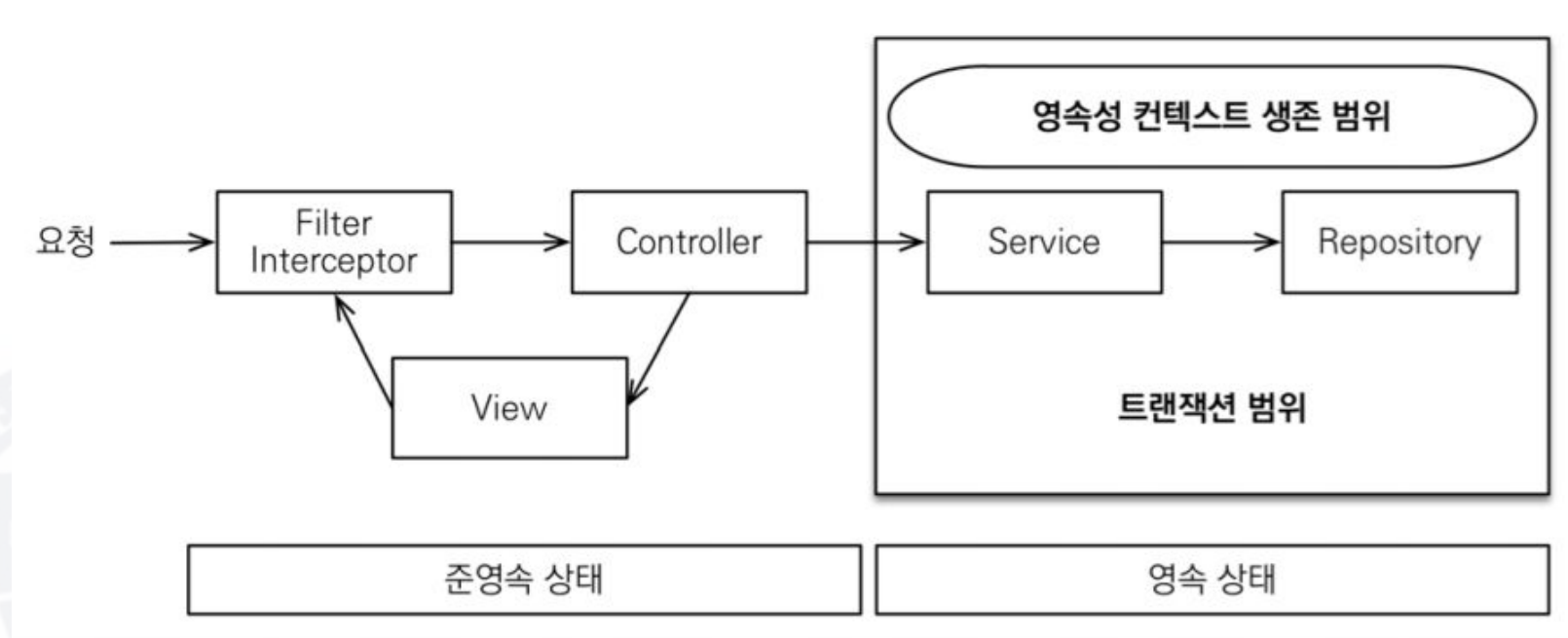
Open Session In View



OSIV ON

- OSIV(Open Session In View)는 영속성 컨텍스트를 뷰까지 열어두는 기능이다. 영속성 컨텍스트가 유지되면 엔티티도 영속 상태로 유지된다.
- **spring boot**에서는 기본적으로 **true**로 활성화
- **view**에서도 영속성 컨텍스트를 사용하므로 데이터베이스 커넥션을 사용하는 라이프 사이클이 길다.

Open Session In View



OSIV OFF

- OSIV를 끄면 트랜잭션을 종료할 때 영속성 컨텍스트를 닫고, 데이터베이스 커넥션도 반환한다. 따라서 커넥션 리소스를 낭비하지 않는다.
- OSIV를 끄면 모든 지연로딩을 트랜잭션 안에서 처리해야 한다.
- **view template**에서 지연로딩이 동작하지 않는다. 트랜잭션이 끝나기 전에 지연 로딩을 강제로 호출해 두어야 한다.

N+1 Problem

연관 관계가 설정된 엔티티를 조회할 경우에 조회된 데이터 갯수(n) 만큼 연관관계의 조회 쿼리가 추가로 발생하여 데이터를 읽어오는 현상

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private long id;
    private String firstName;
    private String lastName;

    @ManyToOne(fetch = FetchType.EAGER)    // 즉시 로딩
    @JoinColumn(name = "team_id", nullable = false)
    private Team team;
}
```

```
@Entity
public class Team {
    @Id
    @GeneratedValue
    private long id;
    private String name;

    @OneToMany(fetch = FetchType.EAGER)
    private List<User> users = new ArrayList<>();
}
```

N+1 Problem

연관 관계가 설정된 엔티티를 조회할 경우에 조회된 데이터 갯수(n) 만큼 연관관계의 조회 쿼리가 추가로 발생하여 데이터를 읽어오는 현상. **Cartesian product** 가 발생하여 **entity** 갯수에 맞게 **subquery** 들이 추가수행

```
List<Team> all = teamRepository.findAll();
System.out.println("===== N+1 시점 확인용
=====");
all.stream().forEach(team -> {
    team.getUsers().size();
});
```

Hibernate: select team0_.id as id1_0_, team0_.name as name2_0_ from team team0_

===== N+1 시점 확인용 =====

Hibernate: select users0_.team_id as team_id1_1_0_, users0_.users_id as users_id2_1_0_, user1_.id as id1_2_1_, user1_.first_name as first_na2_2_1_, user1_.last_name as last_nam3_2_1_, user1_.team_id as team_id4_2_1_ from team_users users0_ inner join user user1_ on users0_.users_id=user1_.id where users0_.team_id=?

Hibernate: select users0_.team_id as team_id1_1_0_, users0_.users_id as users_id2_1_0_, user1_.id as id1_2_1_, user1_.first_name as first_na2_2_1_, user1_.last_name as last_nam3_2_1_, user1_.team_id as team_id4_2_1_ from team_users users0_ inner join user user1_ on users0_.users_id=user1_.id where users0_.team_id=?

Hibernate: select users0_.team_id as team_id1_1_0_, users0_.users_id as users_id2_1_0_, user1_.id as id1_2_1_, user1_.first_name as first_na2_2_1_, user1_.last_name as last_nam3_2_1_, user1_.team_id as team_id4_2_1_ from team_users users0_ inner join user user1_ on users0_.users_id=user1_.id where users0_.team_id=?

Hibernate: select users0_.team_id as team_id1_1_0_, users0_.users_id as users_id2_1_0_, user1_.id as id1_2_1_, user1_.first_name as first_na2_2_1_, user1_.last_name as last_nam3_2_1_, user1_.team_id as team_id4_2_1_ from team_users users0_ inner join user user1_ on users0_.users_id=user1_.id where users0_.team_id=?

N+1 Problem

해결 방안

- **join fetch** : 데이터를 조회할 때, 연관된 데이터도 같이 조회, **inner join**
- **@EntityGraph** : fetch join
- **Batch size** : in 절의 갯수를 제한하여 이슈 방지
- 비정규화

```
public interface TeamRepository extends JpaRepository<Team, Long> {  
    @Query("select t from Team t join fetch t.users")  
    List<Team> findAllFetchJoin();  
}
```

Hibernate: select team0_.id as id1_0_0_, user2_.id as id1_2_1_, team0_.name as name2_0_0_, user2_.first_name as first_na2_2_1_, user2_.last_name as last_nam3_2_1_, user2_.team_id as team_id4_2_1_, users1_.team_id as team_id1_1_0_, users1_.users_id as users_id2_1_0_ from team team0_ inner join team_users users1_ on team0_.id=users1_.team_id inner join user user2_ on users1_.users_id=user2_.id
===== N+1 시점 확인용 =====

Entity 선언, Repository 구현

To DO

- ❑ SQL practice
- ❑ Entity 설계
- ❑ Repository 구현

The background is a solid blue color with several abstract geometric elements. In the top-left and bottom-right corners, there are rectangular areas filled with a pattern of small, light-blue dots. Overlaid on these and the rest of the background are several thin, light-blue lines and circles. A diagonal line runs from the top-left towards the center. Another diagonal line runs from the bottom-right towards the center. There are also two circles: one in the lower-left quadrant and one in the upper-right quadrant. The overall design is modern and minimalist.

EOD