

1. Key (기본키, 후보키, 슈퍼키 등등...) 에 대해 설명해 주세요.

설명

데이터베이스에서 **Key(키)**는 테이블 내의 튜플(레코드)을 유일하게 식별하기 위해 사용되는 중요한 개념입니다.

🔑 1. Super Key (슈퍼키)

- 한 릴레이션(테이블)에서 튜플을 유일하게 식별할 수 있는 속성 또는 속성들의 집합.
- 하나 이상의 속성으로 구성될 수 있으며, 유일성은 보장되지만 최소성은 보장되지 않음.

학생 테이블 (학번, 이름, 이메일) -> 슈퍼키 예시: {학번}, {이메일}, {학번, 이름}, {학번, 이메일}

🔑 2. Candidate Key (후보키)

- 슈퍼키 중에서 속성의 수가 가장 적은, 즉 최소성을 만족하는 키.
- 하나의 테이블에 여러 개의 후보키가 존재할 수 있으며, 이 중 하나를 기본키(**Primary Key**)로 선택함.

학생 테이블에서 {학번}, {이메일}이 각각 유일하다면 둘 다 후보키가 될 수 있음.

🔑 3. Primary Key (기본키)

- 후보키 중에서 주 식별자로 선택된 키.
- **NULL**을 가질 수 없고, 중복도 허용되지 않음.
- 하나의 테이블에는 반드시 하나의 기본키만 존재.

🔑 4. Alternate Key (대체키)

- 후보키들 중에서 기본키로 선택되지 않은 나머지 키들.
- 기본키 외에 유일한 값을 가지는 키로 사용할 수 있음.

🔑 5. Foreign Key (외래키)

- 다른 테이블의 기본키를 참조하는 키.
 - 테이블 간의 관계를 표현할 때 사용.
 - 외래키는 참조하는 기본키의 값만 가질 수 있으며, 데이터 무결성을 유지하는 데 중요함.
-

🔑 6. Composite Key (복합키)

- 두 개 이상의 속성으로 구성된 키로, 이 속성들의 조합이 유일성을 만족해야 함.
 - 보통 단일 속성만으로 유일하지 않을 때 사용.
-

🔑 7. Unique Key

- 중복을 허용하지 않는 키.
- 기본키처럼 유일성을 가지지만, **NULL**을 허용할 수 있음.
- 하나의 테이블에 여러 개 존재할 수 있음.

기본키는 수정이 가능한가요?

1. 기술적으로는 수정 가능

하지만 아래와 같은 문제가 발생할 수 있습니다.

2. 기본키를 수정하는 것이 바람직하지 않은 이유

✅ 1) 참조 무결성 제약 위반 가능

- 다른 테이블에서 외래키(**Foreign Key**)로 해당 기본키를 참조하고 있다면, 기본키를 수정할 경우 참조 관계가 깨지거나 에러가 발생할 수 있습니다. cascade 관련해서

✅ 2) 논리적으로 변경되면 안 되는 값일 가능성

- 기본키는 해당 레코드를 유일하게 식별하는 값입니다.
- 보통 학번, 사번, 주민등록번호, **UUID** 등 변경되지 않는 값을 사용하는 것이 일반적입니다.

✅ 3) 시스템 전체에 영향

- 기본키는 다양한 곳에서 사용될 수 있습니다.
 - 예: 조인, 조회, 외래키, 로그 테이블 등

- 기본키 값이 변경되면 관련된 모든 테이블과 쿼리에서 함께 변경해주어야 하므로 유지보수 비용이 큼.

✓ 클러스터링 인덱스(Clustered Index)

◆ 개념

- 인덱스와 실제 데이터가 같은 **B+Tree** 구조 안에 저장됨
- 즉, 인덱스의 순서 = 실제 레코드의 저장 순서
- MySQL(InnoDB)에서는 기본키가 클러스터링 인덱스
 - ◆ 물리적 저장 방식
- 디스크에 있는 데이터 페이지 자체가 인덱스 구조에 포함됨
- 따라서 인덱스를 따라가면 곧바로 실제 레코드에 도달함 → 추가 조회 없이 바로 사용 가능

◆ 장점

- 범위 검색, 정렬 성능 우수 (인덱스가 곧 정렬된 데이터)
- **I/O 효율이 높음**

◆ 단점

- 기본키 변경/삽입 시 물리적 정렬 필요 → 성능 저하 가능
- 하나의 테이블에 하나만 존재 가능

사실 **MySQL**의 경우, 기본키를 설정하지 않아도 테이블이 만들어집니다. 어떻게 이게 가능한 걸까요?

MySQL에서는 기본키를 명시하지 않아도 테이블을 생성할 수 있습니다. 이는 SQL 표준에서도 기본키를 필수로 요구하지 않기 때문이며, MySQL의 스토리지 엔진인 **InnoDB**의 내부 동작 방식과 관련이 있습니다.

InnoDB는 클러스터형 인덱스를 사용하는 구조입니다. 클러스터형 인덱스란, 테이블의 실제 데이터가 특정 인덱스를 기준으로 정렬되고 저장되는 방식을 의미합니다.

InnoDB는 클러스터 인덱스를 반드시 가져야 하므로, 다음과 같은 우선순위로 이를 결정합니다:

1. 명시된 기본키가 있다면 → 해당 키를 클러스터 인덱스로 사용
2. 기본키가 없고, **NOT NULL + UNIQUE** 제약을 가진 컬럼이 있다면 → 그 컬럼을 클러스터 인덱스로 사용

3. 그마저도 없다면 → InnoDB가 6바이트의 내부 Row ID를 자동 생성하여 클러스터 인덱스로 사용합니다

즉, 기본키가 없어도 InnoDB는 내부적으로 레코드를 유일하게 식별할 수 있는 방법을 항상 갖고 있기 때문에 테이블 생성이 가능한 것입니다.

외래키 값은 NULL이 들어올 수 있나요?

✅ 외래키(Foreign Key) 컬럼에는 NULL 값이 들어올 수 있습니다.

🔍 이유 설명

외래키는 다른 테이블의 기본키 또는 유니크 키를 참조하는 제약 조건입니다. 하지만 SQL 표준과 대부분의 데이터베이스 시스템(MySQL 포함)에서는

외래키 컬럼에 NULL 이 들어가는 것은 제약 조건을 위반하는 것이 아니다
라고 명시하고 있습니다.

🔥 NULL이 지니는 의미

FK에 NULL값이 들어 갈 수 있는지 알아보기 이전에, 먼저 NULL이 지니는 의미에 대해 먼저 알아보겠습니다.

NULL 이 의미하는 바는 다음과 같습니다.

- 1 값이 존재하지 않는다
- 2 값이 존재하지만, 아직 그 값이 무엇인지 알지 못한다
- 3 해당 사항과 관련이 없다

FK는 관계를 맺고자하는 테이블의 PK 값을 참조합니다. 반대로 생각해보면 관계를 맺고자하는 테이블의 PK 값이 이미 존재해야 합니다.

이러한 상황에 관계를 맺고자하는 테이블의 레코드가 아직 생성되지 않았다면?
값이 존재하지만(언젠가 값이 추가될 것이지만), 아직 그 값을 알 수 없다는 의미가 됩니다.
따라서 FK에는 NULL이 들어갈 수 있습니다.

이유는?

- 외래키 제약은 "값이 있는 경우"에만 참조 무결성을 검사합니다.

- **NULL** 은 '값이 없다'는 의미이기 때문에 참조할 필요가 없는 것으로 간주됩니다.

어떤 컬럼의 정의에 **UNIQUE** 키워드가 붙는다고 가정해 봅시다. 이 컬럼을 활용한 쿼리의 성능은 그렇지 않은 것과 비교해서 어떻게 다를까요?

- MySQL에서는 **UNIQUE** 제약이 설정된 컬럼에 대해 자동으로 인덱스를 생성합니다.
- 이 인덱스는 해당 컬럼에 대해 검색, 정렬, 조인 등에서 성능 향상에 기여합니다.
- **UNIQUE** 제약은 인덱스 성능 향상 외에도 중복을 막는 제약 조건이므로 데이터 삽입 시 추가적인 체크 비용이 존재합니다.
- 즉, **INSERT** 나 **UPDATE** 에서 성능 오버헤드가 조금 있을 수 있습니다.
- 인덱스는 성능을 향상시키지만, 남용하면 오히려 쓰기 성능에 악영향을 줄 수 있습니다.

2. RDB와 NoSQL의 차이에 대해 설명해 주세요.

설명

RDB는 테이블 기반의 정형 데이터를 저장하며, 스키마가 고정되어 있고 트랜잭션과 정합성이 강한 것이 특징입니다. 반면 NoSQL은 문서, 키-값, 그래프 등 다양한 형태로 데이터를 저장하며, 유연한 스키마와 수평 확장성에 강점을 가집니다. 따라서 정합성이 중요한 금융 시스템엔 RDB가, 유연성과 확장성이 필요한 SNS나 로그 시스템엔 NoSQL이 적합합니다.

NoSQL의 강점과, 약점이 무엇인가요?

✅ NoSQL의 강점(장점)

1. 📦 스키마 유연성

- 스키마 정의 없이 자유롭게 필드 추가/변경 가능
- 구조가 자주 바뀌는 데이터, 빠른 개발에 유리
- 📌 예: JSON 형태로 자유롭게 저장되는 MongoDB

2. 🌐 수평 확장성(Scale-out) 용이

- 여러 대의 서버에 쉽게 데이터를 분산 저장 가능
- 대량의 데이터를 처리하거나, 트래픽이 높은 환경에 적합
- 📌 예: Cassandra, MongoDB는 분산 저장을 기본 전제로 설계

3. ⚡ 빠른 읽기/쓰기 성능 (특정 조건 하)

- RDB보다 복잡한 JOIN, 트랜잭션 처리를 피하기 때문에 빠름

- 캐시, 로그 저장, 실시간 피드 등에서 성능 우수
- 📌 예: Redis는 메모리 기반 키-값 저장소로 빠른 응답 제공

4. 🧩 다양한 데이터 모델 지원

- 문서형, 키-값형, 그래프형, 컬럼형 등 데이터 특성에 따라 선택 가능
- 📌 예: Neo4j는 복잡한 관계 데이터를 그래프로 표현

❌ NoSQL의 약점(단점)

1. 🔄 JOIN, 정교한 쿼리의 제한

- 일반적으로 JOIN, 복잡한 관계 연산을 지원하지 않거나 성능이 떨어짐
- 📌 관계가 복잡한 데이터에는 부적합

2. 🗑️ ACID 트랜잭션의 제한

- 대부분의 NoSQL은 트랜잭션을 완벽하게 보장하지 않음
- **BASE** 모델: 일관성보다 가용성과 확장성을 우선
- 📌 은행, 회계 시스템 등 강한 정합성 요구 시스템에 부적합

3. ⚠️ 데이터 중복, 정규화 부족

- JOIN이 없기 때문에 데이터를 중복 저장하는 경우 많음
- 유지보수 시 일관성 관리가 어려울 수 있음

4. 🧑🔧 표준화 부족 및 학습 곡선

- 제품별 기능 차이가 커서 사용법이 상이하고, 표준 SQL처럼 통일된 언어가 없음
- 운영 및 마이그레이션이 어려울 수 있음

NoSQL은 분산 시스템을 전제로 설계되어, 데이터 샤딩, 복제, 라우팅 등을 자동으로 처리할 수 있는 구조를 갖고 있습니다. 반면 MySQL은 전통적인 중앙 집중형 구조로, 수평 확장을 위해 복잡한 샤딩 로직이나 복제를 별도로 구현해야 합니다. 또한 NoSQL은 JOIN이나 강력한 트랜잭션을 포기한 대신, 노드 간 동기화 부담이 적어 수평 확장에 매우 유리한 구조입니다.

RDB의 어떠한 특징 때문에 NoSQL에 비해 부하가 많이 걸릴 "수" 있을까요?
(주의: 무조건 NoSQL이 RDB 보다 빠르다라고 생각하면 큰일 납니다!)

1. 🔄 JOIN 연산과 정규화 구조

- RDB는 데이터 중복을 줄이기 위해 정규화를 적용합니다.
- 이로 인해 실무에서는 많은 테이블을 **JOIN**하여 데이터를 조합하는 쿼리가 많습니다.
- 테이블이 커질수록 JOIN 연산 비용이 기하급수적으로 커지며,
- 디스크 **I/O**, **CPU** 부하 증가로 이어질 수 있습니다.

1. 🗝️ 강한 일관성을 위한 트랜잭션 처리 (ACID)

- RDB는 트랜잭션에서 원자성, 일관성, 고립성, 지속성을 보장하기 위해
→ 락(Lock), MVCC, 로그 기록 등 다양한 내부 처리를 수행합니다.
- 위와 같은 트랜잭션은 내부적으로 락을 걸어 동시성 제어를 하고,
- 이는 다중 사용자가 동시에 접근할 때 성능 저하나 대기 시간 증가로 이어질 수 있습니다.

1. 🛠️ 스키마 고정 및 구조 변경 비용

- RDB는 스키마가 고정되어 있어서 `ALTER TABLE` 같은 구조 변경 시 전체 테이블 재작성이 필요할 수 있음
- 데이터량이 많을 경우 이러한 작업은 운영 환경에 큰 부하를 유발할 수 있음

1. 📈 수직 확장에 의존

- RDB는 전통적으로 수직 확장(Scale-up) 중심
- 하드웨어 성능의 한계에 도달하면 병목이 발생하고, 확장 비용도 매우 큼
- 만약 데이터를 여러 서버에 분산해버리면 JOIN은 서버 간 통신 비용이 발생하고 비효율적이 됩니다.
- 그래서 단일 서버에서 모든 데이터를 처리하는 수직 확장 구조에 의존하게 됩니다.

NoSQL을 활용한 경험이 있나요? 있다면, 왜 RDB를 선택하지 않고 해당 DB를 선택했는지 설명해 주세요.

네, Redis를 사용한 경험이 있습니다.

이전 프로젝트에서 게시물의 하루 조회수와 총 조회수를 저장하고 조회하는 기능을 구현할 때 Redis를 활용했습니다.

Redis는 인메모리 기반의 키-값 저장소로 읽기/쓰기 속도가 매우 빠르기 때문

✅ RDB 대신 Redis를 선택한 이유

Redis를 선택한 이유 중 하나는 바로 `INCR` 명령어 덕분이었습니다.

하루 조회수나 총 조회수를 증가시킬 때, **Redis**에서는 `INCR` 하나로 원자적(**atomic**)이고 빠르게 처리할 수 있기 때문에,
매 요청마다 조회수를 처리해도 병목 없이 동작했습니다.

반면 RDB를 사용할 경우에는 다음과 같은 처리 흐름이 필요합니다:

1. 현재 값을 `SELECT`

2. 값을 증가시켜 `UPDATE`

이 과정에서 동시 요청이 많을 경우에는 락(**Lock**) 처리, 또는 트랜잭션 처리가 필요하게 되어 부하가 발생할 수 있습니다.

Redis는 `INCR` 명령어 자체가 원자적으로 동작하기 때문에
추가적인 동기화 코드 없이도 안전하게 동시성 처리가 가능했고,
이는 성능뿐만 아니라 개발 생산성 측면에서도 유리했습니다.

"빠른 처리 속도 + 원자적 연산 지원(**INCR**) + 동시성 안전성 확보"

이 세 가지가 RDB 대신 Redis를 선택한 이유입니다.

이 경험을 통해 Redis가 단순한 캐시 외에도 실시간 수치 집계나 임시 저장소로도 매우 유용하다는 것을 체감했고,
시스템 특성과 성능 요구에 따라 적절한 기술을 선택하는 것이 중요하다는 점을 배웠습니다.

3. 트랜잭션이 무엇이고, ACID 원칙에 대해 설명해 주세요.

설명

1. 트랜잭션(Transaction)이란?

하나의 작업 단위를 의미하며, 데이터베이스에서 반드시 모두 성공하거나 모두 실패해야 하는 작업 집합입니다.

- 예를 들어, 송금 같은 작업은 다음과 같은 단계를 거칩니다:
 1. A 계좌에서 1만 원 차감
 2. B 계좌에 1만 원 추가
 - 이 두 작업은 한 번에 처리되어야 하며, 한 쪽만 실행되면 안 됩니다.
→ 이처럼 여러 데이터 변경을 하나의 논리적 단위로 묶는 것이 트랜잭션입니다.
-

✅ 2. ACID 원칙이란?

트랜잭션이 제대로 작동하기 위해 데이터베이스는 **ACID** 원칙을 지켜야 합니다.
ACID는 다음 네 가지 속성의 약자입니다.

◆ A — Atomicity (원자성)

- 트랜잭션의 작업은 모두 성공하거나, 모두 실패해야 함
- 중간에 오류가 나면 전체 작업이 롤백(Rollback)됨

◆ C — Consistency (일관성)

- 트랜잭션 수행 전과 수행 후의 데이터 상태는 항상 일관되어야 함
- 데이터 제약조건, 규칙 등이 항상 유지되어야 함
- 계좌 금액이 마이너스가 되지 않는 조건, 외래키가 존재하는 경우만 insert 등

◆ I — Isolation (고립성)

- 동시에 여러 트랜잭션이 실행되더라도, 서로 간섭받지 않아야 함
- A 트랜잭션이 주문 테이블을 수정 중일 때,
 - B 트랜잭션이 중간 상태의 데이터를 조회해서는 안 됨
- 💡 격리 수준(Isolation Level)과 관련
- 고립성을 구현하는 수준에는 단계가 있음 (트레이드오프 있음)
 - READ UNCOMMITTED
 - READ COMMITTED
 - REPEATABLE READ (MySQL InnoDB 기본)
 - SERIALIZABLE

고립성이 높을수록 정합성은 좋지만 성능 저하 발생

◆ D — Durability (지속성)

- 트랜잭션이 성공적으로 완료되면, 그 결과는 영구적으로 저장됨
- 시스템이 꺼져도 데이터는 보존됨
- DBMS는 트랜잭션을 커밋할 때, 데이터를 디스크에 반영하고 로그로 기록함
- 일반적으로 **redo log** 또는 **write-ahead logging (WAL)** 사용

ACID 원칙 중, Durability를 DBMS는 어떻게 보장하나요?

디스크 기록이 핵심이다

RAM은 전원이 꺼지면 내용이 사라지므로

→ 커밋된 데이터는 반드시 디스크에 기록되어야 안전합니다.

그래서 DBMS는 트랜잭션 커밋 시 다음 순서를 따릅니다:

1. 트랜잭션 변경 내용을 로그(**redo log**, **WAL**)에 기록
2. 로그가 디스크에 성공적으로 **flush** 됐는지 확인
3. 이후에만 커밋을 완료

즉, 트랜잭션 커밋이 성공했다는 것은
그 변경사항이 디스크에 안전하게 기록됐다는 뜻

2. **Write-Ahead Logging (WAL)**

- 대부분의 DBMS(MySQL InnoDB, PostgreSQL 등)는 **WAL** 또는 **Redo log** 방식을 사용합니다.
- WAL의 핵심 원칙:

"데이터를 디스크에 반영하기 전에, 변경 로그를 먼저 디스크에 기록하라"

- 로그만 살아 있으면, 시스템이 꺼져도 다시 복구할 수 있음
- ### 2. 장애 발생 시 복구 방식
- 시스템 장애 후 재시작 시,
DBMS는 디스크에 남은 로그를 읽어들여

- 아직 반영되지 않은 로그 → 다시 적용 (Redo)
- 실패한 트랜잭션 → 무시 (Rollback)

InnoDB는 트랜잭션의 지속성을 보장하기 위해 **Redo Log(After Image)**를 사용하고,
트랜잭션 중 오류 발생 시 롤백을 위해 **Undo Log(Before Image)**도 함께 유지합니다.

Write-Ahead Logging 원칙에 따라, 변경 내용을 실제 데이터에 쓰기 전에 **Redo Log**를 디스크에 먼저 기록하고,

COMMIT은 로그가 디스크에 안전하게 flush된 뒤에 완료됩니다.

이를 통해 장애가 발생해도 커밋된 트랜잭션은 **Redo Log**로 복구,
미커밋 트랜잭션은 Undo Log로 롤백이 가능하게 됩니다.

✓ 흐름 요약

1. 트랜잭션이 데이터 변경 요청
2. **Undo Log**에 이전 값 저장 (롤백 대비)
3. 변경 내용을 **Redo Log**에 기록 (**After Image**)
4. COMMIT 시 **Redo Log**가 디스크에 flush됨
5. 실제 데이터는 나중에 반영되더라도 OK
6. 장애 시 → Redo Log를 이용해 복구(**Replay**), Undo Log로 롤백

트랜잭션을 사용해 본 경험이 있나요? 어떤 경우에 사용할 수 있나요?

네, 트랜잭션을 사용한 경험이 있습니다.

이전에 제가 개발한 서비스에서 "유저가 질문에 대답하면 포인트를 얻는 기능"을 구현한 적이 있는데,

이 과정에서 여러 작업을 하나의 트랜잭션으로 묶어 처리했습니다.

- 질문에 대한 유저의 답변을 저장
- 해당 유저의 포인트를 증가

만약 답변은 저장됐지만 포인트는 지급되지 않는다면, 사용자 경험에 혼란을 줄 수 있고
반대로 포인트만 올라가고 답변이 저장되지 않는다면 시스템적으로 오류가 발생합니다.

✅ 트랜잭션 도입 이유

이런 이유로 이 세 작업을 하나의 트랜잭션으로 묶어서 처리했고,
도중에 하나라도 실패하면 전체 작업을 롤백(Rollback) 하도록 구현했습니다.

읽기에는 트랜잭션을 걸지 않아도 될까요?

◆ InnoDB는 MVCC를 사용합니다 (멀티 버전 동시성 제어)

- 트랜잭션이 없어도, **SELECT**는 커밋된 최신 버전의 데이터를 읽음
- 기본 격리 수준인 **REPEATABLE READ**에서는 트랜잭션 내에서는
→ 항상 트랜잭션 시작 시점의 스냅샷을 기준으로 읽음

📌 즉, 트랜잭션을 시작해야만 "일관된 스냅샷"을 보장받을 수 있음

✅ 트랜잭션 없이 SELECT 할 경우

```
SELECT * FROM posts WHERE id = 1;
```

- 기본적으로 가장 최근에 커밋된 값을 읽음
- 읽는 도중 다른 트랜잭션이 값을 변경하거나 커밋하면 → 다음 SELECT에선 다른 값이 보일 수 있음

✅ 트랜잭션 안에서 SELECT 할 경우

```
START TRANSACTION; SELECT * FROM posts WHERE id = 1; -- 이후 이 행이  
수정되거나 커밋돼도 내 트랜잭션에서는 변하지 않음 SELECT * FROM posts WHERE id  
= 1; COMMIT;
```

- 같은 트랜잭션 안에서는 반복해서 읽어도 값이 동일 → "Repeatable Read"
- 트랜잭션이 끝나기 전까지는 스냅샷이 고정
- ✅ 그래서 언제 SELECT에 트랜잭션이 필요할까?

상황	트랜잭션 필요 여부	설명
단순 조회 (read only)	❌ 불필요	최신 데이터 조회면 충분
반복 조회 시 결과 동일해야 함	✅ 필요	트랜잭션 내 스냅샷 필요
읽고 나서 조건에 따라 UPDATE/INSERT	✅ 필요	동시성 문제 방지
집계 쿼리, 보고서 생성	✅ 필요	중간에 데이터가 바뀌면 안 됨

4. 트랜잭션 격리 레벨에 대해 설명해 주세요.

설명:

1. READ UNCOMMITTED

- 가장 낮은 수준, 성능은 가장 빠름
- 다른 트랜잭션이 아직 커밋하지 않은 데이터도 읽을 수 있음
예시
- 트랜잭션 A: `UPDATE post SET views = 100 WHERE id = 1`
- 트랜잭션 B: `SELECT views FROM post WHERE id = 1` → 아직 A가 COMMIT 안 했지만, B는 100을 읽을 수 있음 (Dirty Read)

◆ 2. READ COMMITTED

- 커밋된 데이터만 읽을 수 있음 (Dirty Read 방지)
- 하지만 같은 쿼리를 반복 실행해도 결과가 달라질 수 있음 (Non-repeatable Read)
예시
- 트랜잭션 A: `SELECT views FROM post WHERE id = 1` → 10
- 트랜잭션 B: `UPDATE post SET views = 20 WHERE id = 1; COMMIT;`
- 트랜잭션 A: 다시 `SELECT` → 20
→ 같은 트랜잭션 내에서도 읽는 값이 바뀜

◆ 3. REPEATABLE READ (MySQL InnoDB 기본)

- 반복해서 읽는 값이 항상 동일
- 하지만 조건을 만족하는 행 수가 바뀌는 경우는 허용 (Phantom Read)
예시
- 트랜잭션 A: `SELECT * FROM orders WHERE status = 'WAITING'`
- 트랜잭션 B: `INSERT INTO orders(status) VALUES ('WAITING'); COMMIT;`
- 트랜잭션 A가 다시 `SELECT` 하면 → 새 행이 보일 수 있음 (Phantom Read)

4. SERIALIZABLE

- 가장 높은 격리 수준 → 모든 트랜잭션이 마치 *순차적으로 실행되는 것처럼 보장
- 모든 SELECT에 공유 락을 걸어 동시성 성능은 급격히 저하
예시
- 트랜잭션 A: `SELECT * FROM post WHERE views > 100`
- 트랜잭션 B: 이 조건을 만족하는 데이터를 INSERT하려고 하면 → A의 트랜잭션이 끝날 때까지 대기 or 실패

MySQL은 **Gap Lock**을 통해 Phantom Read도 방지하려고 시도하기 때문에, 실무에서는 REPEATABLE READ에서도 거의 안전한 편입니다.

모든 **DBMS**가 4개의 레벨을 모두 구현하고 있나요? 그렇지 않다면 그 이유는 무엇일까요?

✅ 결론부터 말씀드리면:

❌ 모든 **DBMS**가 **SQL** 표준의 4가지 트랜잭션 격리 수준을 모두 지원하는 것은 아닙니다.

그리고 표준과 동일한 이름의 수준을 지원한다고 해도, 구현 방식과 동작은 **DBMS**마다 다를 수 있습니다.

1. ⚙️ **DBMS**마다 동시성 제어 방식이 다르기 때문

- DBMS마다 내부적으로 사용하는 동시성 제어 메커니즘이 다릅니다:
 - 예) **MVCC**, 락 기반(**Locking**), 타임스탬프 기반, 낙관적 동시성 제어 등
- 이러한 차이로 인해 격리 수준을 완전히 표준대로 구현하는 것이 어려운 경우가 있음

1. 표준은 추상적이지만, 구현은 구체적이어야 함

- SQL 표준은 "이런 일이 발생해서는 안 된다"는 개념적인 정의만 제시합니다.
 - 예: Non-repeatable Read가 없어야 한다
- 반면 실제 구현은

- 어떤 락을 언제 잡고,
- 어느 시점에 커밋을 허용할지,
- 어떤 성능 저하를 감수할지를 **DBMS 개발자가 직접 선택**해야 합니다

→ 따라서 어떤 DBMS는 격리 수준을 부분적으로만 지원하거나,
같은 이름의 수준이라도 동작 방식이 서로 다를 수 있습니다.

실제 예시

◆ MySQL InnoDB

- 4가지 격리 수준 이름은 모두 지원
- 하지만 기본값인 **REPEATABLE READ**는 **Gap Lock**과 **Next-Key Lock**을 사용해
→ **Phantom Read**까지 방지하려 시도
→ 즉, **SQL** 표준의 의미보다 더 강력한 구현

◆ PostgreSQL

- **READ UNCOMMITTED**는 사실상 **READ COMMITTED**와 동일하게 동작함
→ 이유: PostgreSQL의 MVCC 구조상 Dirty Read 자체가 발생하지 않음

◆ Oracle

- **READ COMMITTED**와 **SERIALIZABLE**만 명시적으로 제공
- 나머지 수준은 사용자가 직접 제어하거나 지원하지 않음

만약 **MySQL**을 사용하고 있다면, (**InnoDB** 기준) **Undo** 영역과 **Redo** 영역
에 대해 설명해 주세요.

✅ 1. Undo 영역 (Undo Log)

📌 목적

트랜잭션 도중 문제가 발생했을 때 롤백(Rollback) 하기 위해 사용

📌 주요 역할

- 트랜잭션이 완료되기 전까지 변경 이전의 데이터를 버퍼 풀(메모리)에 저장
- 트랜잭션이 롤백될 경우, Undo Log를 참고하여 변경을 되돌림
- 동시에 실행되는 트랜잭션에게는 **MVCC** 기반의 스냅샷 읽기를 제공하기 위한 버전 관리 기능도 수행
저장 위치
- 주로 버퍼 풀 내 **Undo Segment**에 저장
- 필요 시 디스크에도 저장

2. Redo 영역 (Redo Log)

📌 목적

장애가 발생했을 때, 트랜잭션 결과를 재적용(Redo) 하여 지속성(Durability)을 보장

📌 주요 역할


- 변경된 데이터를 디스크에 즉시 반영하지 않고,
Redo Log에 먼저 기록한 뒤 나중에 반영
- 시스템이 장애로 종료되어도, **Redo Log**를 보고 커밋된 데이터를 복구

그런데, 스토리지 엔진이 정확히 무엇을 하는 건가요?

한 줄 정의

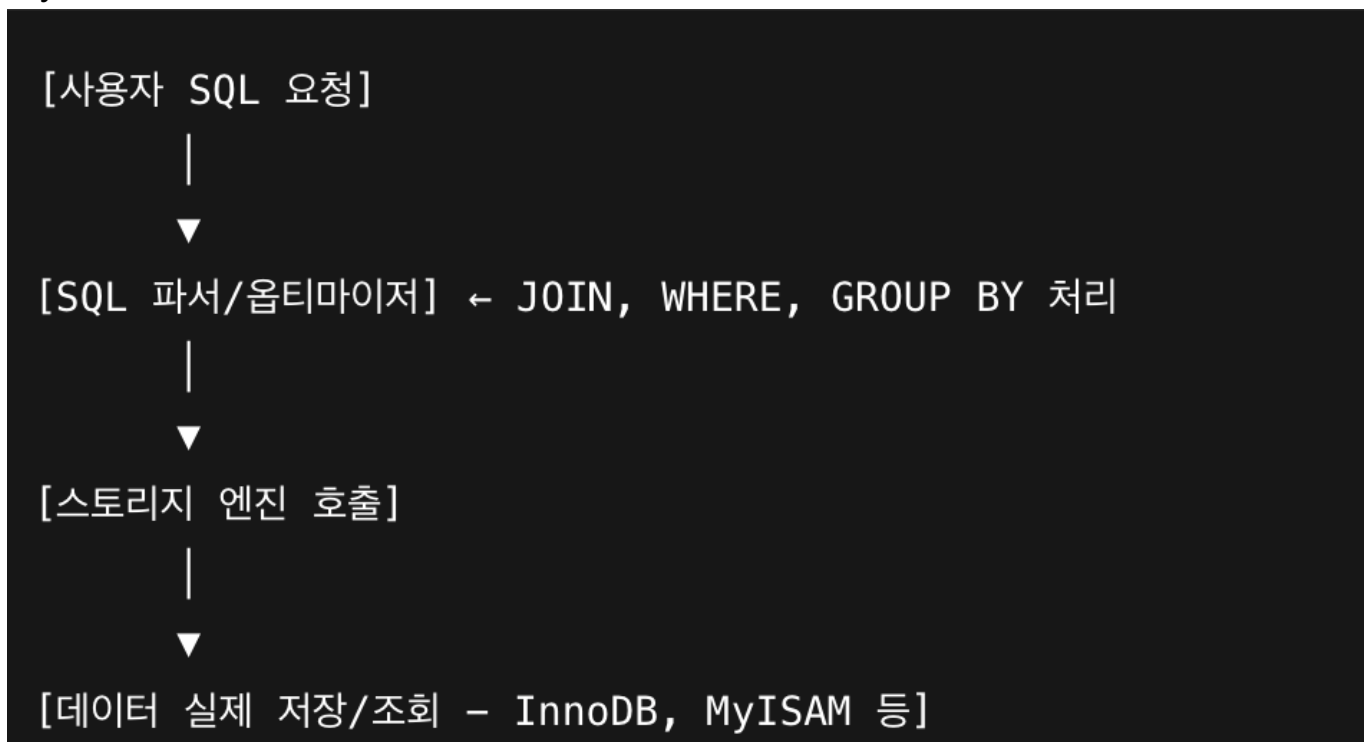
스토리지 엔진은 **MySQL**이 데이터를 디스크에 저장하고,
검색하고, 수정하고, 삭제하는 방법을 구현한 모듈입니다.

스토리지 엔진의 역할 요약

기능	설명
 데이터 저장 방식 정의	데이터를 어떤 형식으로, 어떤 파일에 저장할지 결정
 읽기/쓰기 처리	SELECT, INSERT, UPDATE, DELETE 쿼리를 실제로 수행
 트랜잭션 지원 여부 결정	InnoDB는 트랜잭션 지원, MyISAM은 지원 X
 락 처리 방식 결정	행 단위 락 vs 테이블 단위 락
 인덱스 구현 방식	B+ Tree, Hash 등 사용 방식이 다름
 복구 및 로그 관리	Redo/Undo log 사용 여부, 충돌 복구 기능 등

MySQL 아키텍처에서의 위치

MySQL은 **SQL** 계층과 스토리지 계층이 분리되어 있습니다:



- **SQL** 처리와 데이터 저장 로직을 분리해 두었기 때문에,
- 다양한 스토리지 엔진을 플러그인처럼 교체 가능하게 만든 구조입니다.

예: 동일한 SQL이라도 엔진마다 다르게 동작

→ InnoDB로 만들면:

- 트랜잭션 가능
- 외래키 설정 가능
- 충돌 시 자동 복구 가능

```
CREATE TABLE posts ( id INT PRIMARY KEY, title VARCHAR(100) )
ENGINE=MyISAM;
```

→ MyISAM으로 만들면:

- 트랜잭션 불가능
- 외래키 불가능
- 충돌 시 수동 복구 필요

5. 인덱스가 무엇이고, 언제 사용하는지 설명해 주세요.

설명

한 줄 정의

인덱스는 데이터베이스에서 원하는 데이터를 빠르게 찾기 위해 사용하는 자료구조입니다. 마치 책의 목차처럼, 전체 데이터를 일일이 확인하지 않고 원하는 행(row)을 빠르게 조회할 수 있도록 도와줍니다.

인덱스가 필요한 이유

- 데이터가 많아질수록 조건에 맞는 데이터를 찾는 데 시간이 오래 걸림
- 인덱스는 특정 컬럼 값을 기준으로 정렬된 구조를 유지하여,
→ 검색 속도를 획기적으로 향상시킴

언제 인덱스를 사용하는가?

1. **WHERE** 조건절에서 자주 검색되는 컬럼

```
SELECT * FROM users WHERE username = 'hong';
```

2. **JOIN** 대상이 되는 컬럼

```
SELECT * FROM orders o JOIN users u ON o.user_id = u.id;
```

3. 정렬 (ORDER BY), 그룹핑 (GROUP BY), DISTINCT

```
SELECT * FROM products ORDER BY price;
```

4. LIKE 검색 (접두사 매칭)

일반적으로 인덱스는 수정이 잦은 테이블에선 사용하지 않기를 권합니다. 왜 그럴까요?

인덱스는 데이터를 검색할 때는 빠르지만, 데이터를 수정할 때는 성능 오버헤드가 발생하기 때문입니다.

특히 수정이 자주 일어나는 테이블에서는 인덱스 유지 비용이 커져서 전체 성능을 떨어뜨릴 수 있습니다.

1. ⚙ 쓰기 연산 시 인덱스도 함께 수정되어야 함

- INSERT/UPDATE/DELETE가 발생할 때,
해당 테이블의 모든 관련 인덱스도 함께 갱신해야 합니다.
- 예를 들어, `username`, `email`, `created_at` 에 각각 인덱스가 있다면:

```
UPDATE user SET email = 'new@ex.com' WHERE id = 1;
```

→ `email` 컬럼뿐만 아니라

→ **email** 인덱스의 구조도 변경되어야 함 (B+ Tree 재정렬 등)

2. 🔄 인덱스 수정은 비용이 큰 작업

- 인덱스는 보통 **B+ Tree**로 구성되어 있습니다.
- 중간에 삽입/삭제가 발생하면 노드 분할(split), 병합(merge), 재조정(rebalance)이 일어날 수 있음
- → 추가적인 디스크 I/O와 CPU 자원 소모 발생

3. 📉 쓰기 성능 저하

- 데이터 자체는 빨리 쓰이지만,
인덱스도 함께 쓰이면서 디스크 I/O 병목이 발생
- 인덱스가 많을수록 → 쓰기 성능이 눈에 띄게 감소

4. 📁 인덱스가 많으면 저장 공간도 증가

- 각 인덱스는 추가적인 저장 공간을 사용하므로
데이터 변경이 많을수록 인덱스 유지 비용이 누적

앞 꼬리질문에 대해, 그렇다면 인덱스에서 사용하지 않겠다고 선택한 값은 위 정책을 그대로 따라가나요?

❌ **DBMS**는 사용자가 “인덱스를 사용하지 않겠다”고 명시하지 않는 이상, 가능한 인덱스가 있다면 실행 계획(**Execution Plan**)에 따라 자유롭게 인덱스를 사용할 수 있습니다.

즉, “인덱스를 만들지 않았다면, 당연히 사용할 수 없고”,
“인덱스를 만들어 두었지만 사용하지 않게 하고 싶다면, 따로 힌트나 제한을 줘야 합니다.”

IGNORE INDEX (idx_username)

FORCE INDEX (idx_created_at)

❌ 수정할 때 인덱스를 사용하지 않겠다고 '선언'할 수는 없습니다.

왜냐하면,

인덱스는 단순히 조회용이 아니라, 무결성 제약을 지키고 인덱스 자체를 유지하기 위해
DBMS가 반드시 사용해야 하기 때문입니다.

🔧 인덱스를 직접 무시하고 수정하는 방법은 없다

- MySQL이나 대부분의 RDBMS에서는 **UPDATE** 또는 **DELETE** 시 사용자가 특정 인덱스를 강제로 무시하는 기능은 제공하지 않습니다.
- **IGNORE INDEX**, **FORCE INDEX** 는 **SELECT** 문에서만 사용 가능하며, **UPDATE** 나 **DELETE** 에서는 단순히 **WHERE** 조건의 탐색 경로를 제한할 뿐, 인덱스 자체의 유지/갱신은 반드시 수행됩니다.
인덱스를 사용하지 않겠다고 하면 어떤 문제가 생기나?

예: **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY** 제약이 붙은 인덱스

- **UPDATE** 시 값을 변경하려면,
해당 인덱스를 확인해서 중복 여부를 검사해야 합니다

- 이걸 무조건 인덱스를 사용해야만 가능한 작업입니다
(즉, 무결성을 위한 내부 강제 탐색)

✅ 요점:

인덱스를 사용하지 않고도 데이터를 수정할 수 있는 방법은 없고,
인덱스가 존재하는 한 **DBMS**는 내부적으로 반드시 인덱스를 갱신합니다.

ORDER BY/GROUP BY 연산의 동작 과정을 인덱스의 존재여부와 연관지어 설명해 주세요.

✅ 인덱스가 존재하면 정렬/그룹핑을 "인덱스를 따라가면서" 처리할 수 있기 때문에 성능이 빠릅니다.

❌ 인덱스가 없으면 전체 데이터를 정렬하거나 그룹핑해야 하므로 비용이 큼.

ORDER BY의 동작 방식과 인덱스

1. 💎 인덱스가 있는 경우 (인덱스 정렬)

```
SELECT * FROM users ORDER BY created_at;
```

- `created_at`에 인덱스가 있다면:
 - 인덱스 자체가 이미 정렬된 **B+ Tree** 구조이므로
 - 정렬 연산 없이 인덱스를 따라가면서 데이터만 가져오면 됨
 - → **Using index** 또는 **Index Range Scan** 수행

✅ 매우 빠름 (추가 정렬 비용 없음)

❌ 인덱스가 없는 경우

- DB는 모든 데이터를 읽은 뒤, 메모리나 디스크에 정렬 작업을 따로 수행
- → **Using filesort** 라는 실행 계획이 뜸 (실제로는 파일이 아닐 수도 있음)

! 이 경우 성능 저하 가능성 ↑

✅ GROUP BY의 동작 방식과 인덱스

1. 💎 인덱스가 있는 경우

```
SELECT status, COUNT(*) FROM orders GROUP BY status;
```

- `status` 컬럼에 인덱스가 있다면:
 - 인덱스를 통해 같은 값을 순차적으로 읽기 때문에
 - → 그룹을 빠르게 구분 가능
 - → **Using index for group-by** 또는 **index scan + grouping** 으로 처리
- ✓ 빠른 그룹핑 가능

2. ✗ 인덱스가 없는 경우

- 모든 데이터를 읽고 메모리 상에서 해시나 정렬 기반 그룹핑 수행
 - 데이터 양이 많을 경우 → 임시 테이블 사용 + 디스크 쓰기 발생
- ! 성능 저하 가능성 ↑

기본키는 인덱스라고 할 수 있을까요? 그렇지 않다면, 인덱스와 기본키는 어떤 차이가 있나요?

◆ 기본키는 무조건 인덱스를 포함합니다 → ✓ “기본키는 인덱스를 포함한다”는 말은 맞습니다.

✗ 하지만 인덱스가 곧 기본키는 아니며,
기본키는 단순 인덱스 이상으로 무결성 제약을 포함한 개념입니다.

기본키는 테이블에서 각 행(row)을 유일하게 식별하기 위한 컬럼(또는 컬럼 조합)입니다.

기본키는 무결성 보장(중복 불가, NULL 불가)을 위한 제약 조건이고,

인덱스는 데이터를 빠르게 검색하기 위한 자료구조입니다.

기본키를 설정하면 내부적으로 유니크 인덱스가 자동 생성되므로 기본키는 인덱스를 포함한다고 할 수 있지만,

반대로 단순한 인덱스가 기본키 역할을 할 수는 없습니다.

특히 **InnoDB**에서는 기본키가 클러스터링 인덱스**로 동작하며, 테이블 저장 구조까지 영향을 주는 중요한 역할을 합니다.

MySQL InnoDB에서의 특별한 점: 클러스터링 인덱스

- InnoDB는 기본키를 클러스터링 인덱스로 사용합니다
- 테이블의 실제 데이터가 기본키 순서대로 정렬되어 저장됨
- → 기본키 = 인덱스이자 물리적인 저장 순서 기준

먼저, 클러스터링 인덱스란?

데이터 행(Row) 자체가 인덱스 구조 안에 포함된 인덱스

→ 즉, **B+ Tree** 인덱스의 **leaf** 노드가 "기본키 + 전체 행 데이터"로 구성됨

📌 그래서 클러스터링 인덱스 = 인덱스 + 실제 데이터

기본키가 클러스터링 인덱스로 쓰이는 이유

◆ 1. 모든 데이터가 기본키 기준으로 정렬됨

- 데이터가 기본키 순서로 디스크에 물리적으로 저장됨
- → `ORDER BY id`, `BETWEEN`, `>` 등 기본키 기반 범위 조회가 빠름

◆ 2. 데이터 접근 시 테이블과 인덱스가 분리되지 않음

- 일반 인덱스(보조 인덱스)는 인덱스 → 테이블 **Row Pointer**(기본키) → 실제 데이터
⇒ 두 번 접근 (인덱스 → 테이블)
- 클러스터링 인덱스는
인덱스를 타고 내려가면 곧바로 데이터가 있음 → 한 번 접근

InnoDB는 왜 기본키를 클러스터링 인덱스로 고정할까?

이유	설명
✅ 기본키는 항상 존재하고	테이블마다 유일함 + NULL 불가 → 안정적 기준
✅ 중복되지 않음	중복되면 정렬이 불가능해짐
✅ 가장 자주 사용됨	JOIN, WHERE 절, PK 조회 등에서 가장 많이 쓰임
✅ 물리적 저장 정렬 기준으로 적합	B+ Tree 구조 유지에 적합한 속성

✅ 읽기 성능 향상, I/O 비용 감소

그렇다면 외래키는요?

외래키는 한 테이블의 컬럼이 다른 테이블의 기본키 또는 유니크 키를 참조하도록 설정하는 제약 조건입니다.

즉, 테이블 간의 관계를 연결해주는 키입니다.

외래키를 통해 두 테이블 사이의 관계를 논리적으로 표현하고, 데이터 정합성을 자동으로 유지할 수 있습니다.

외래키는 상대 테이블의 기본키나 유니크 키를 참조해야 하며, 이들에게는 이미 유니크 인덱스가 자동으로 생성되어 있으므로 별도로 인덱스를 만들 필요는 없습니다.

다만 외래키를 가진 자기 테이블 쪽 컬럼에는 참조 무결성을 빠르게 검사하기 위해 인덱스가 필요하며,

MySQL에서는 이 인덱스를 자동으로 생성하거나 없으면 에러가 발생하기도 합니다.

인덱스가 데이터의 물리적 저장에도 영향을 미치나요? 그렇지 않다면, 데이터는 어떤 순서로 물리적으로 저장되나요?

결론부터 말씀드리면:

✅ MySQL의 InnoDB 스토리지 엔진에서는 “기본키에 의한 클러스터링 인덱스”가 데이터의 물리적 저장 순서를 결정합니다.

❌ 일반적인 보조 인덱스는 물리적 저장 순서에 영향을 미치지 않습니다.

클러스터링 인덱스 (기본키 기준)

- InnoDB는 테이블 데이터를 기본키 순서대로 **B+ Tree** 구조에 저장합니다.
- 즉, 기본키 인덱스의 리프 노드에 실제 데이터 전체가 들어 있음
- 이것을 **클러스터형(Clustered Index)**라고 부릅니다.

보조 인덱스 (Secondary Index)

- 예: `CREATE INDEX idx_email ON users(email);`
- 보조 인덱스는 **email** 컬럼만 정렬된 인덱스 트리를 따로 유지합니다.
- 리프 노드는 해당 row의 기본키 값만 저장하고,
실제 전체 데이터는 클러스터링 인덱스를 따라가야 찾을 수 있음 (→ 인덱스 루프업)

MyISAM과는 다르다

- MyISAM은 클러스터링 인덱스가 없고,
→ 데이터는 삽입된 순서대로 저장됨 → 인덱스는 데이터 위치(파일 오프셋)를 가리키는 역할만 함

우리가 아는 **RDB**가 아닌 **NoSQL** (ex. Redis, MongoDB 등)는 인덱스를 갖고 있나요? 만약 있다면, **RDB**의 인덱스와는 어떤 차이가 있을까요?

결론 요약

- ✅ 네, Redis, MongoDB 등 대부분의 NoSQL 시스템도 '인덱스'를 가지고 있습니다.
! 하지만 인덱스의 구조와 역할, 생성 방식은 RDB와 다릅니다.

1. Redis의 인덱스 개념

- ◆ Redis는 "인덱스"라는 개념이 명시적으로는 없습니다.
하지만 실질적으로는 **키(key)** 자체가 RDB의 인덱스 역할을 합니다.

✅ 구조적으로 보면:

- Redis는 **key-value** 구조
- 모든 데이터는 **key** 기준으로 정렬된 자료구조(해시 테이블, 트리 등)에 저장
- 따라서 `GET key`, `EXISTS key`, `DEL key` 등은 $O(1)$ 에 가까운 속도로 동작
📌 별도로 인덱스를 만들 필요 없이, 키 자체가 곧 빠른 탐색 수단

! 단점:

- value 내부 필드에 대해서는 직접적인 인덱싱 불가능
- 복잡한 검색이나 범위 조건에 맞는 조회에는 적합하지 않음

2. MongoDB의 인덱스

- ◆ MongoDB는 **RDB**와 유사한 인덱스 시스템을 제공합니다.

- 기본적으로 `_id` 필드에 자동 인덱스가 생성됨 (PRIMARY KEY처럼)
- 사용자 정의 인덱스도 가능:

✓ MongoDB 인덱스의 특징:

항목	내용
구조	B+ Tree 기반 인덱스 사용 (RDB와 유사)
복합 인덱스	{ name: 1, age: -1 } 가능
텍스트 인덱스	"text" 인덱스로 단어 기반 검색 지원
지리공간 인덱스	2D/Geo 인덱스로 위치 기반 쿼리 지원
다중 인덱스	하나의 필드에 여러 인덱스 종류 가능 (복합, 텍스트 등)

✓ RDB 인덱스와의 차이점

항목	RDB 인덱스	NoSQL 인덱스 (MongoDB/Redis 등)
목적	정형 데이터의 정렬/검색 최적화	다양한 구조의 유연한 검색 최적화
자료구조	주로 B+ Tree	MongoDB: B+ Tree, Redis: 해시 테이블, skiplist 등
기본 인덱스	PRIMARY KEY → 클러스터링 인덱스	Redis: key 자체, MongoDB: <code>_id</code> 필드
자동화 수준	제약조건과 강하게 연동됨 (PK, UNIQUE 등)	제약은 약하고, 인덱스는 명시적으로 생성해야 함
정렬과 조인	인덱스로 정렬/조인 최적화	조인 없음 / 정렬은 일부 제한적 지원

✓ 정리하면

- **Redis:** key 자체가 인덱스 역할, value 내부는 직접 인덱싱 불가
- **MongoDB:** 인덱스 구조는 RDB와 매우 유사하지만, 스키마가 자유롭고 다양한 인덱스 옵션이 있음
- **RDB:** 무결성 제약과 강한 연관성, 고정된 스키마 기반에서 인덱스를 설계

MongoDB도 RDB처럼 B+ Tree 기반의 인덱스를 사용하지만, 유연한 문서 구조에 맞춰 다양한 인덱스 종류(텍스트, 지리공간, 배열 등)를 제공한다는 점이

가장 큰 차이입니다.

RDB의 인덱스는 무결성 제약(PK/UNIQUE)과 밀접하게 연관되어 있지만, MongoDB는 인덱스를 단순 성능 최적화 도구로 분리해서 사용합니다.

또한 RDB는 테이블 기반의 정형화된 인덱싱을, MongoDB는 문서 내부까지 포함한 비정형 필드의 인덱싱까지 유연하게 처리합니다.

(A, B) 와 같은 방식으로 인덱스를 설정한 테이블에서, A 조건 없이 B 조건만 사용하여 쿼리를 요청했습니다. 해당 쿼리는 인덱스를 탈까요?

일반적으로 복합 인덱스는 선두 컬럼부터 조건이 있어야 인덱스를 탈 수 있지만, 선두 컬럼의 카디널리티가 낮고, 후속 컬럼 조건이 선택적일 경우, MySQL 8.0 이상에서는 **Index Skip Scan**을 통해 선두 조건 없이도 인덱스를 활용할 수 있습니다.

이 방식은 선두 컬럼의 가능한 값을 하나씩 고정하며 후속 컬럼을 탐색하는 방식으로, 범위 스캔은 아니지만 전체 테이블 스캔보다는 훨씬 효율적입니다.

6. RDBMS, NoSQL에서의 클러스터링/레플리케이션 방식에 대해 설명해 주세요.

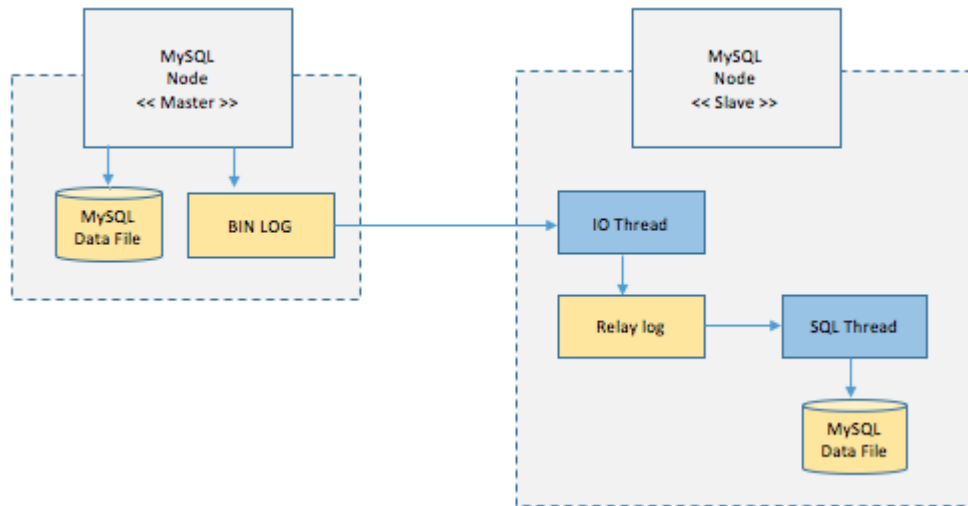
설명

1. 리플리케이션(Replication)이란?

리플리케이션(Replication)이란?

리플리케이션이란 여러 개의 DB를 권한에 따라 수직적인 구조(Master-Slave)로 구축하는 방식이다. 리플리케이션에서 Master Node는 쓰기 작업 만을 처리하며 Slave Node는 읽기 작업 만을 처리한다. 리플리케이션은 비동기 방식으로 노드들 간의 데이터를 동기화하는데, 자세한 처리 방법은 아래와 같다.

리플리케이션(Replication) 처리 방식



위의 그림은 MySQL의 Replication 방식에 대한 그림이며 자세한 처리 순서는 아래와 같다.

1. Master 노드에 쓰기 트랜잭션이 수행된다.
2. Master 노드는 데이터를 저장하고 트랜잭션에 대한 로그를 파일에 기록한다.(BIN LOG)
3. Slave 노드의 IO Thread는 Master 노드의 로그 파일(BIN LOG)를 파일(Replay Log)에 복사한다.
4. Slave 노드의 SQL Thread는 파일(Replay Log)를 한 줄씩 읽으며 데이터를 저장한다.

리플리케이션은 Master와 Slave간의 데이터 무결성 검사(데이터가 일치하는지)를 하지 않는 비동기방식으로 데이터를 동기화한다. 이러한 구조에 의해 리플리케이션 방식은 다음과 같은 장점과 단점을 갖고 있다.

리플리케이션(Replication) 장점과 단점

- 장점
 - DB 요청의 60~80% 정도가 읽기 작업이기 때문에 Replication만으로도 충분히 성능을 높일 수 있다.
 - 비동기 방식으로 운영되어 지연 시간이 거의 없다.
- 단점
 - 노드들 간의 데이터 동기화가 보장되지 않아 일관성있는 데이터를 얻지 못할 수 있다.

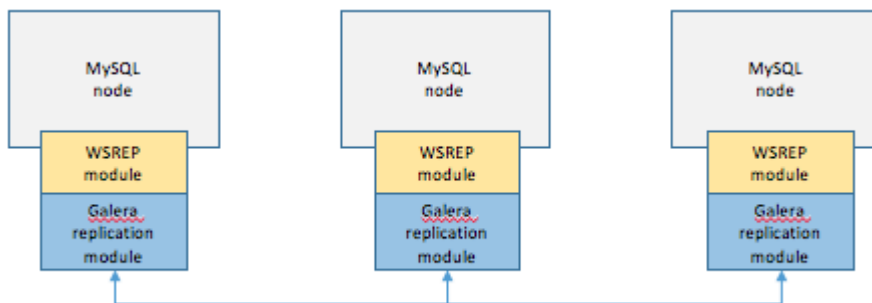
- Master 노드가 다운되면 복구 및 대처가 까다롭다.

2. 클러스터링(Clustering)이란?

클러스터링(Clustering)이란?

클러스터링이란 여러 개의 DB를 수평적인 구조로 구축하는 방식이다. 클러스터링은 분산 환경을 구성하여 Single point of failure와 같은 문제를 해결할 수 있는 Fail Over 시스템을 구축하기 위해서 사용된다. 클러스터링은 동기 방식으로 노드들 간의 데이터를 동기화하는데, 자세한 처리 방법은 아래와 같다.

클러스터링(Clustering) 처리 방식



위의 그림은 MySQL의 Clustering 방식 中 Galera 방식에 대한 그림이며 자세한 처리 순서는 아래와 같다.

1. 1개의 노드에 쓰기 트랜잭션이 수행되고, COMMIT을 실행한다.
2. 실제 디스크에 내용을 쓰기 전에 다른 노드로 데이터의 복제를 요청한다.
3. 다른 노드에서 복제 요청을 수락했다는 신호(OK)를 보내고, 디스크에 쓰기를 시작한다.
4. 다른 노드로부터 신호(OK)를 받으면 실제 디스크에 데이터를 저장한다.

클러스터링은 DB들 간의 데이터 무결성 검사(데이터가 일치하는지)를 하는 동기방식으로 데이터를 동기화한다. 이러한 구조에 의해 클러스터링 방식은 다음과 같은 장점과 단점을 갖고 있다.

클러스터링(Clustering) 장점과 단점

- 장점
 - 노드들 간의 데이터를 동기화하여 항상 일관성있는 데이터를 얻을 수 있다.

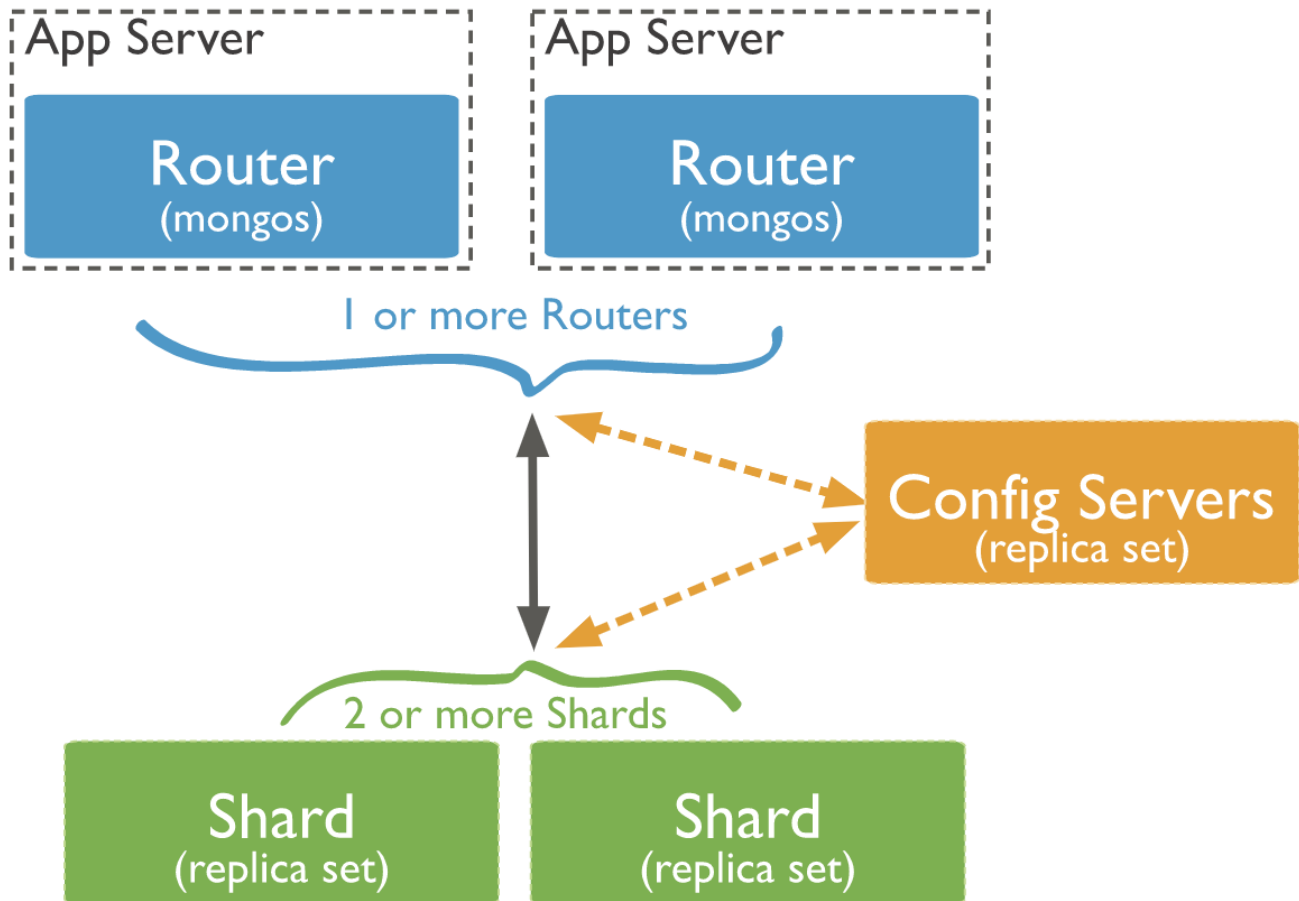
- 1개의 노드가 죽어도 다른 노드가 살아 있어 시스템을 계속 장애없이 운영할 수 있다.
- 단점
 - 여러 노드들 간의 데이터를 동기화하는 시간이 필요하므로 Replication에 비해 쓰기 성능이 떨어진다.
 - 장애가 전파된 경우 처리가 까다로우며, 데이터 동기화에 의해 스케일링에 한계가 있다.

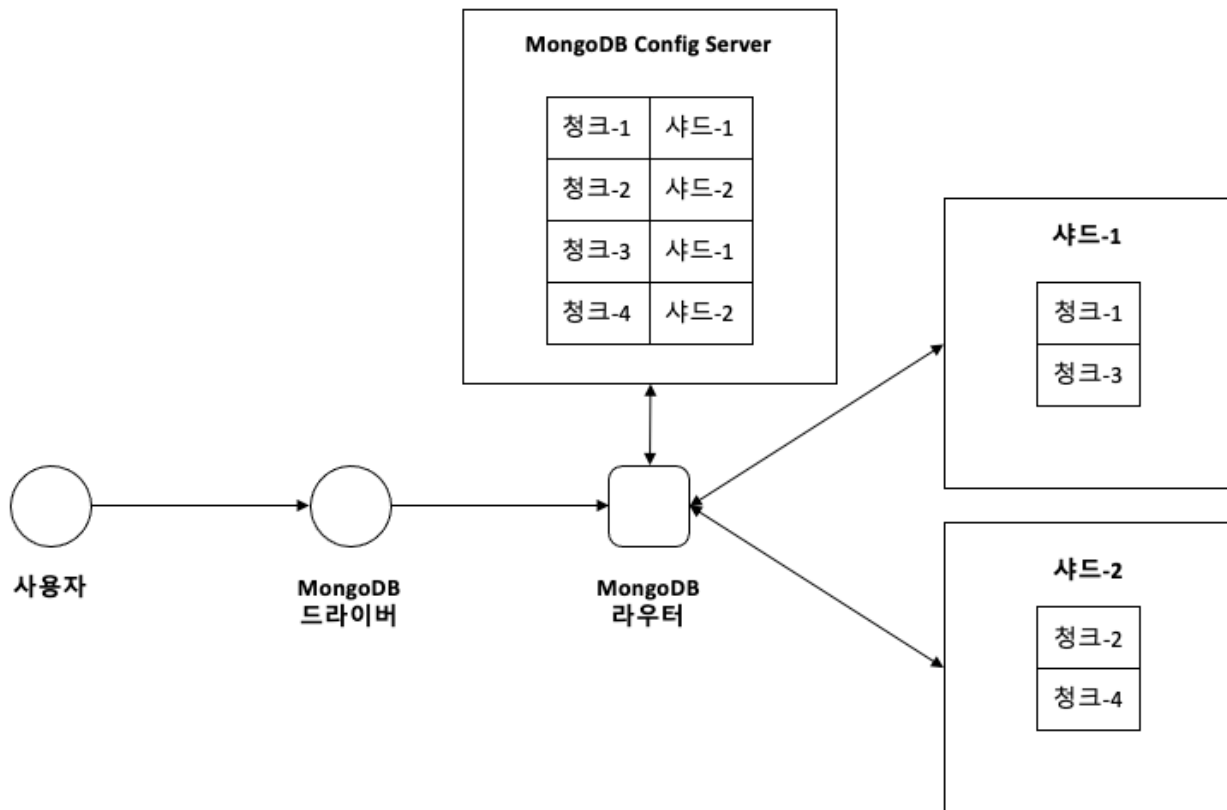
클러스터링을 구현하는 방법으로는 또 Active-Active와 Active-Standby가 있다. Active-Active는 클러스터를 항상 가동하여 가용가능한 상태로 두는 구성 방식이고, Active-Standby는 일부 클러스터는 가동하고, 일부 클러스터는 대기 상태로 구성하는 방식이다. 상황에 따라 알맞은 구조를 선택하면 된다.

MongoDB 사용 시(NoSQL)

[clustering]

Sharded cluster





동작 과정

처리과정

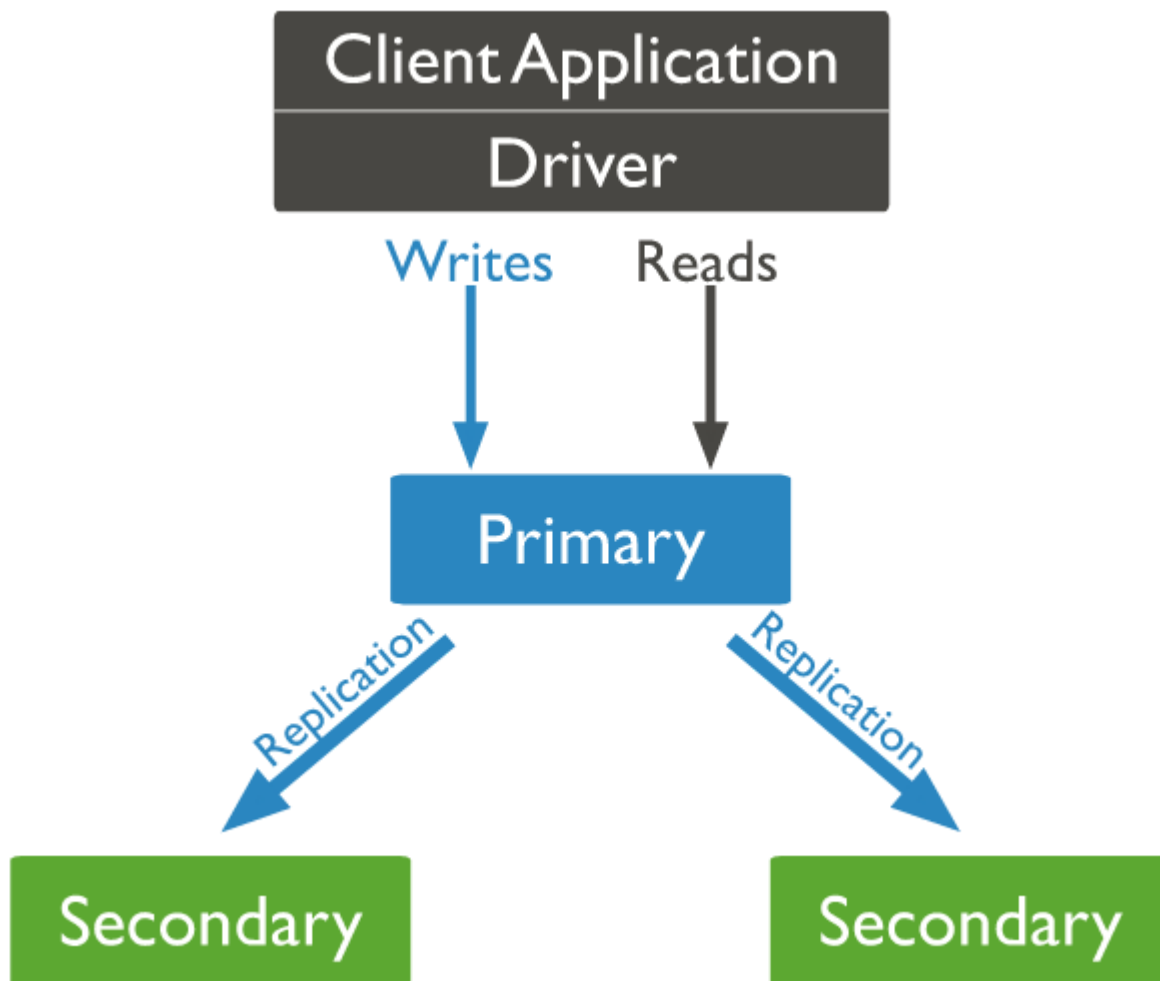
- 쿼리가 참조하는 컬렉션의 Chunk metadata를 Config Server로부터 가져와 router의 메모리에 캐시한다.
- 쿼리의 조건에서 Sharding Key 조건을 찾는다.
- 1) Sharding Key 존재할 경우 : 해당 Sharding Key가 포함된 Chunk 정보를 router의 캐시에 검색하여 Shard서버로만 사용자 쿼리를 요청
- 2) Sharding Key 없을 경우 : 모든 Shard 서버로 쿼리를 요청한다.
- 쿼리를 전송한 대상 Shard 서버로부터 쿼리 결과가 도착하면 결과를 병합하여 사용자에게 결과를 반환한다.

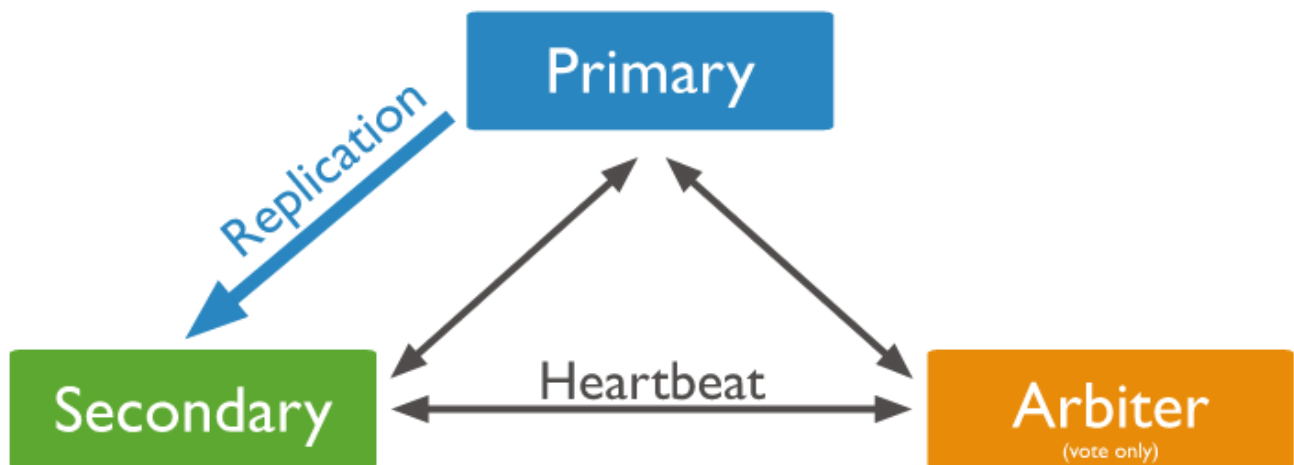
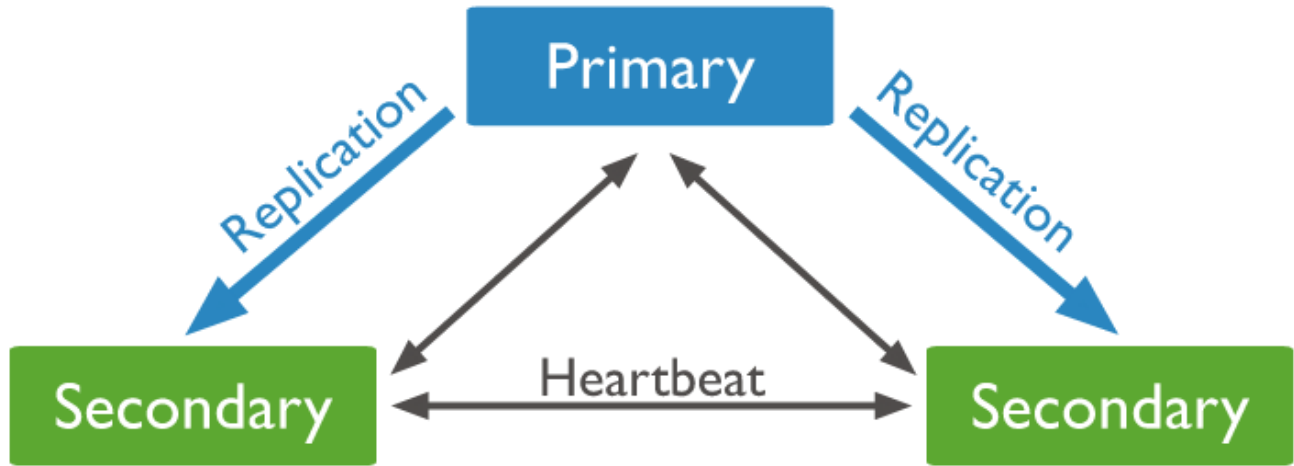
- Mongodb는 직접 특정 Shard에 접근할 수 없음
- Query Router에 명령을 하고, Query Router가 Shard에 접근하는 방식(Config Server정보기반으로 data chunk 위치를 찾아가는 것도 이때 수행됨)
 - **query router** : 쿼리를 받아 각 샤드로 보내주는 역할, 데이터 저장되어 있지 않고 **router** 역할만 수행

- **shard** : 실제 데이터가 저장되는 저장소
- **config** : 어떤 shard가 어떤 데이터(data chunk)를 가지고 있는지, data chunk 들을 어떻게 분산해서 저장하며 관리하러 지 알 수 있음
- 성능 문제를 위해 shard 여러개를 두고 분산처리
 - scaling을 통해 늘리고 줄일 수 있음
 - 보통 3개의 shard 구성(SPOF막기위해)
- Query Router는 Shard정보를 찾는 부분의 성능을 위해 **Config Server**의 **metadata**를 **cache**로 저장해둔다.
 - metadata : 데이터가 저장되어 있는 shard 정보 및 sharding key 정보

[replicaion]

replica-set





- **Primary node / Secondary node** 로 구성
 - Primary node : 모든 쓰기 작업 수행, 기본적으로 읽기 작업도 Primary 몫
- 노드 간 **heartbeat**을 통해 상태체크
- Primary node 사용할 수 없는 경우(장애나 네트워크 이슈) 적절한 Secondary node 는 새로운 Primary 노드 선택을 위한 투표 개최 => 홀수 노드 구성이 좋다
 - 짝수로 구성하게 되면 홀수로 구성한 경우와 다르지 않아 서버 낭비로 이어짐, 쿼럼(Quorum) 구성이 어려울 수 있음
- Arbiter 모드 : Primary node 선출을 위한 투표만 참여, 디스크 저장X, 하나 이상 필요 X

MongoDB의 Replica Set에서 Primary는 모든 쓰기 요청을 처리하고, 기본적으로 읽기도 Primary에서 처리하여 일관성을 보장합니다.

반면 Secondary는 Primary의 데이터를 실시간 복제하여 **Failover**에 대비하며, 설정에 따라 읽기 요청도 분산 처리할 수 있어 조회 부하를 줄이는 데 사용됩니다. Arbiter는 데이터는 저장하지 않고, **Primary** 선출을 위한 투표만 참여하는 경량 노드로, 장애 복구의 안정성을 높이기 위해 사용됩니다.

이러한 분산 환경에선, 트랜잭션을 어떻게 관리할 수 있을까요?

✅ 1. 레플리케이션 환경에서 트랜잭션 처리 (Master-Slave)

구조 요약:

◆ 트랜잭션 처리 방식

- 트랜잭션은 **Master**에서만 처리됩니다.
- Master가 커밋하면 binlog로 기록되고, **Slave**는 이 binlog를 순차적으로 적용
- **Slave**는 트랜잭션을 "재생(replay)"하는 입장

결과:

항목	설명
ACID 보장	오직 Master에서 보장됨
Slave 지연 가능성	복제 지연 → 최신 데이터가 아닐 수 있음
트랜잭션 강도	Master에서는 완전한 ACID, Slave는 Eventually Consistent

✅ 즉, 트랜잭션은 **Master**에서만 보장되며, **Slave**는 단지 데이터를 따라가는 구조입니다.

✅ 2. 클러스터링 환경에서 트랜잭션 처리 (예: Galera Cluster)

구조 요약:

- 모든 노드가 동등한 권한을 갖는 **Multi-Master** 구조
- 각 노드에서 트랜잭션을 수행할 수 있음 (Active-Active)
- 트랜잭션 커밋 시 다른 노드와 동기 복제 수행
 - ◆ 트랜잭션 처리 방식 (Galera 예시)

1. 노드 A에서 트랜잭션 수행 후 COMMIT 요청
2. COMMIT 전에 다른 노드로 복제 요청 전송
3. 모든 노드가 복제 준비 완료(OK 응답) 시 → 실제 커밋 진행

◆ 결과:

항목	설명
ACID 보장	모든 노드에서 보장됨 (동기 복제)
Failover 가능	어떤 노드에서든 쓰기 가능 (Active-Active)
쓰기 성능	노드 수가 많을수록 동기화 비용 발생 → 쓰기 성능 저하 가능

✅ Galera 같은 클러스터링 구조는 노드 간 동기 트랜잭션 복제를 통해 전체적인 일관성과 트랜잭션 안정성을 유지합니다.

✅ 2. 샤딩 환경에서의 트랜잭션 처리 방식

샤딩 구조에서는 데이터가 여러 DB(샤드)에 나눠 저장되기 때문에 하나의 트랜잭션이 여러 샤드를 건너는 경우, 트랜잭션(ACID) 보장이 어려워집니다.

✅ 트랜잭션 처리 방식 ①: 샤드 단위 트랜잭션 (로컬 트랜잭션)

- 한 샤드 안에서의 트랜잭션만 보장됨
- ACID 완전 보장 가능 (단일 노드이므로)

예: `user_id % 4 = 0`인 유저는 `shard_0`에만 있음 → 그 안에서는 안전하게 트랜잭션 가능

✅ 트랜잭션 처리 방식 ②: 분산 트랜잭션 (Cross-Shard Transaction)

- 하나의 트랜잭션이 여러 샤드에 동시에 걸치는 경우
- ACID 보장을 위해 2PC(2단계 커밋) 같은 기술 필요

📌 예: `order` 테이블은 샤드 A, `inventory` 는 샤드 B

```
BEGIN; -- 샤드 A에서 주문 생성 -- 샤드 B에서 재고 차감 COMMIT;
```

→ 이럴 경우, 샤드 간 트랜잭션 조정이 필요

✅ 대표적인 처리 방식:

방식	설명
2PC (2-Phase Commit)	모든 샤드에 사전 준비 요청 → 모두 OK면 커밋
SAGA 패턴	분산 트랜잭션 대신 보상 트랜잭션으로 복구
Eventual Consistency	실시간 동기화 대신 시간이 지나면 동기화되도록 허용

✅ 샤딩에서 트랜잭션 전략 선택 기준

조건	전략
대부분 단일 샤드에서 처리됨	로컬 트랜잭션 사용 (속도 빠름, 구현 간단)
다수 샤드를 자주 넘나드는 구조	2PC, SAGA 필요 → 성능 저하 감수
데이터 일관성보다 확장성과 속도 우선	Eventual Consistency + 보상 트랜잭션 활용

마스터, 슬레이브 데이터 동기화 전까지의 데이터 정합성을 지키는 방법은 무엇이 있을까요?

상황 이해: Master-Slave (Primary-Replica) 구조

- **Master:** 모든 쓰기(**INSERT/UPDATE/DELETE**) 작업 수행
- **Slave:** 읽기 전용, Master의 `binlog` 를 복제받아 동기화
- 대부분은 비동기 또는 반동기 복제
 - ✅ 빠르지만
 - ! **Master**에서의 변경이 **Slave**에 반영되기까지 지연 발생 가능

✅ 문제: 정합성(C) 위배 가능성

사용자가 Master에서 데이터를 갱신한 직후,
Slave에서 같은 데이터를 조회하면 **이전 값이 조회될 수 있음**
→ 특히 읽기 요청을 **Slave로 분산했을 때 발생**

✅ 그럼 어떻게 정합성을 지킬 수 있을까?

✅ 1. **Read-after-write consistency** 보장

쓰기 후 바로 읽는 요청은 반드시 **Master에서** 처리하도록 제한

방법:

- 로그인, 글 작성, 결제 등 즉시 반영되어야 하는 요청은 무조건 **Master로**
- 캐시 또는 Proxy, 라우터 수준에서 제어

POST /user/123/profile → Master 사용 GET /user/123/profile → 최근 수정
된 사용자면 Master로 라우팅

✅ 사용자 체감 일관성 보장

! Master 부하 집중 가능성 있음

✅ 2. **Semi-synchronous Replication** (반동기 복제)

Master가 트랜잭션을 커밋할 때, 최소한 **1개의 Slave**가 복제를 완료했다는 응답을 받은
뒤 커밋 완료

📌 MySQL, PostgreSQL 등에서 지원

- 동기보단 빠르고, 비동기보단 안정적
- 트랜잭션 손실 방지, 지연 최소화

✅ 정합성 ↑

! 쓰기 속도는 다소 저하됨

✅ 3. **GTID 기반 트랜잭션 추적 (MySQL 기준)**

Global Transaction ID로 Master와 Slave의 트랜잭션 동기화 상태를 정확히 추적할 수
있음

- 사용자 요청에서 마지막 커밋된 **GTID**를 기억
- 이후 Slave의 GTID가 해당 값보다 작다면 → Slave가 최신 상태 아님 → Master로 라우팅

✓ 정확한 트랜잭션 동기 상태 확인 가능

! 복잡한 구현 필요

✓ 4. 읽기 일관성 옵션 활용

일부 DBMS에서는 클라이언트가 일관성 수준을 지정 가능

- DynamoDB: `ConsistentRead = true`
- MongoDB: `readConcern = "majority"`
- MySQL: `group_replication_consistency` 설정

✓ 읽기 일관성 요구 수준 조절 가능

! 성능과 일관성의 트레이드오프 필요

✓ 5. 쓰기 후 일정 시간 캐싱 / 지연 조회 (soft delay)

변경 직후 민감한 데이터는 캐시를 통해 일정 시간 **Master** 기준 유지

- 예: 게시글 수정 후, 5초간 캐시 고정
- 또는, 변경된 사용자는 일정 시간 Slave가 아닌 Master로만 조회

✓ 변경 직후 일관성 유지

! 실시간성 요구가 낮을 때만 가능

✓ 전략 종합 요약

전략	설명	장점	단점
쓰기 후 Master 에서 읽기	Write-Read 일관성 보장	정합성 확실	Master 부하 집중
반동기 복제 (Semi-sync)	복제 완료 전까지 Commit 지연	안전성 향상	쓰기 속도 느려질 수 있음

전략	설명	장점	단점
GTID 기반 요청 분기	동기화 상태 실시간 판단	정확한 판단	구현 복잡
읽기 일관성 옵션 사용	DB의 일관성 레벨 설정	선택적 일관성 확보	성능 손실 가능
캐싱/지연 전환	일정 시간 동안만 Master 참조	간단	완전한 일관성은 아님

💬 면접용 요약 멘트

Master-Slave 복제 환경에서는 복제 지연으로 인해 정합성 문제가 발생할 수 있습니다. 이를 해결하기 위해 쓰기 후 읽기는 반드시 **Master**에서 처리(**Read-after-write consistency**)하거나,

Semi-synchronous 복제, **GTID** 기반 요청 분기, 읽기 일관성 옵션 활용 등의 전략이 사용됩니다.

상황에 따라 성능과 정합성의 트레이드오프를 고려하여 적절한 방법을 선택하는 것이 중요합니다.

다중 트랜잭션 상황에서의 **Deadlock** 상황과, 이를 해결하기 위한 방법에 대해 설명해 주세요.

Deadlock(교착 상태)란?

둘 이상의 트랜잭션이 서로가 점유한 자원을 요청하며 무한 대기하는 상태
→ 결국 모두가 아무 작업도 못 하는 상황

📌 간단한 정의:

- 트랜잭션 A가 트랜잭션 B가 점유한 자원을 기다리고 있고,
- 트랜잭션 B는 A가 점유한 자원을 기다리는 상황

→ 🔄 상호 대기 발생 → **Deadlock**

✅ Deadlock 해결 방법

✅ 1. 트랜잭션에서 테이블/행 접근 순서를 통일

트랜잭션마다 자원을 접근하는 순서를 일관되게 유지

✅ Deadlock 원인 중 하나인 순환 대기(Circular Wait) 제거

예:

- 모든 트랜잭션이 항상 계좌 1 → 계좌 2 순서로만 UPDATE
 - 반대로 접근하지 않도록 규칙 설정
- ✅ 2. 트랜잭션 범위를 최소화

트랜잭션은 가능한 한 짧고 단순하게 유지

✅ 자원 점유 시간 줄이기 → Deadlock 발생 가능성 감소

✅ 3. 잠금 강도 줄이기 (Lock Granularity 최소화)

SELECT ... FOR UPDATE 나 UPDATE 로 인한 행 잠금을
→ 필요한 최소 범위로만 설정

✅ 더 적은 자원이 잠기므로 교차 대기 줄어듦

✅ 4. 타임아웃 설정

```
SET innodb_lock_wait_timeout = 5;
```

✅ 지정 시간 안에 락 획득 못하면 → 트랜잭션 자동 실패 → Deadlock 방지

✅ 5. 재시도 로직 구현 (애플리케이션 레벨)

Deadlock 발생 시 트랜잭션을 다시 시도할 수 있도록 로직 구성

- 일반적으로 RDBMS는 Deadlock 발생 시 트랜잭션 중 하나를 rollback시키므로,
- 애플리케이션에서 해당 트랜잭션을 감지하고 자동 재시도 가능

샤딩 방식은 무엇인가요? 만약 본인이 DB를 분산해서 관리해야 한다면, 레플리케이션 방식과 샤딩 방식 중 어떤 것을 사용할 것 같나요?

샤딩(Sharding)이란?

샤딩은 데이터를 여러 DB 서버(Shard)에 나눠서 분산 저장하는 방식입니다.
즉, 하나의 거대한 테이블을 여러 개로 쪼개고, 서버별로 일부만 저장하도록 하는 것

📌 핵심 개념 요약

항목	내용
샤드(Shard)	데이터의 일부를 저장하는 독립된 DB 인스턴스
샤딩 키(Sharding Key)	어떤 데이터를 어떤 샤드에 저장할지 결정하는 기준 값
샤딩 목적	DB의 수평 확장 → 성능 향상, 용량 분산

✅ 샤딩의 종류

방식	설명
수평 샤딩 (Horizontal Sharding)	같은 테이블의 행(Row)들을 나눔 예: <code>user_id % 3 == 0</code> → Shard 1
수직 샤딩 (Vertical Sharding)	테이블을 기능/도메인 단위로 분리 예: <code>User</code> , <code>Order</code> , <code>Product</code> 각 DB로 나눔
하이브리드 샤딩	수직 + 수평을 혼합해서 사용

✅ 샤딩의 장점과 단점

장점	단점
✅ 수평 확장성 우수 (데이터/부하 분산)	❌ 샤딩 키 설계 어려움
✅ 성능 향상 (데이터량 분산)	❌ JOIN, 트랜잭션 어려움 (Cross-Shard Query)
✅ 고용량 처리 가능	❌ 샤드 간 재분배 어려움 (Rebalancing 복잡)

레플리케이션 vs 샤딩 비교

항목	레플리케이션	샤딩
목적	읽기 부하 분산, 고가용성 확보	쓰기/데이터량 분산, 수평 확장
구성	Master → Slave (복제)	데이터 분산 저장 (Shards)
확장성	❌ 수직 확장(한계 있음)	✅ 수평 확장 (확장 용이)
데이터 일관성	Master에서만 보장	샤드 간 일관성 따로 유지
트랜잭션 처리	비교적 쉬움	어렵거나 불가능 (2PC 등 필요)
실무 예	읽기 많은 서비스, 캐시 중심	사용자 수 많은 서비스, 대용량 처리

질문: "어떤 상황에서 샤딩을 선택하겠습니까?"

샤딩은 하나의 테이블 데이터를 여러 개의 **DB**로 수평 분산 저장하는 방식으로, 대용량 데이터를 다루거나 트래픽이 급격히 증가하는 시스템에서 수평 확장성과 성능 향상을 위해 사용됩니다.

반면, 레플리케이션은 하나의 Master DB에서 데이터를 관리하고, 이를 여러 Slave DB에 복제함으로써 읽기 부하 분산과 고가용성을 확보할 수 있습니다. 만약 제가 직접 분산 설계를 해야 한다면, 쓰기 부하가 많고 데이터 크기가 크면 샤딩, 읽기 위주의 트래픽이나 장애 대비 목적이라면 레플리케이션을 선택하겠습니다.

✅ 2. 샤딩 환경에서의 트랜잭션 처리 방식

샤딩 구조에서는 데이터가 여러 DB(샤드)에 나눠 저장되기 때문에 하나의 트랜잭션이 여러 샤드를 건너는 경우, 트랜잭션(**ACID**) 보장이 어려워집니다.

✅ 트랜잭션 처리 방식 ①: 샤드 단위 트랜잭션 (로컬 트랜잭션)

- 한 샤드 안에서의 트랜잭션만 보장됨
- ACID 완전 보장 가능 (단일 노드이므로)

예: `user_id % 4 = 0`인 유저는 `shard_0`에만 있음 → 그 안에서는 안전하게 트랜잭션 가능

✅ 트랜잭션 처리 방식 ②: 분산 트랜잭션 (Cross-Shard Transaction)

- 하나의 트랜잭션이 여러 샤드에 동시에 걸치는 경우
- ACID 보장을 위해 2PC(2단계 커밋) 같은 기술 필요

📌 예: `order` 테이블은 샤드 A, `inventory` 는 샤드 B

```
BEGIN; -- 샤드 A에서 주문 생성 -- 샤드 B에서 재고 차감 COMMIT;
```

→ 이럴 경우, 샤드 간 트랜잭션 조정이 필요

✅ 대표적인 처리 방식:

방식	설명
2PC (2-Phase Commit)	모든 샤드에 사전 준비 요청 → 모두 OK면 커밋
SAGA 패턴	분산 트랜잭션 대신 보상 트랜잭션으로 복구
Eventual Consistency	실시간 동기화 대신 시간이 지나면 동기화되도록 허용

✅ 샤딩에서 트랜잭션 전략 선택 기준

조건	전략
대부분 단일 샤드에서 처리됨	로컬 트랜잭션 사용 (속도 빠름, 구현 간단)
다수 샤드를 자주 넘나드는 구조	2PC, SAGA 필요 → 성능 저하 감수
데이터 일관성보다 확장성과 속도 우선	Eventual Consistency + 보상 트랜잭션 활용

7. 정규화가 무엇인가요?

설명

정규화(Normalization)란?

정규화는 관계형 데이터베이스에서 데이터의 중복을 제거하고,
데이터 무결성을 유지하며, 논리적으로 일관된 구조를 만들기 위한 과정입니다.

→ 하나의 테이블을 여러 개로 나누고,
서로 관계를 맺도록 설계하는 과정

✅ 왜 정규화를 하나요?

목적	설명
✅ 중복 데이터 제거	같은 데이터를 여러 번 저장하지 않음
✅ 삽입/삭제/갱신 이상(Anomaly) 방지	데이터 무결성 보장
✅ 데이터 일관성 유지	잘못된 참조를 막음
✅ 저장 공간 효율화	중복 데이터로 인한 낭비 방지

✅ 정규화의 단계

정규화는 여러 단계로 나뉘며, 보통 **3차 정규형(3NF)**까지 적용하는 것이 일반적입니다.

◆ 1NF (제1정규형)

원자값(**Atomic value**)만 저장 (더 이상 나눌 수 없는 값)

◆ 2NF (제2정규형)

부분 함수 종속 제거 (기본키의 일부에만 종속된 속성 제거)

- 테이블의 기본키가 복합키일 경우,
→ 컬럼이 키 전체에 종속되지 않고 일부에만 종속되면 다른 테이블로 분리

◆ 3NF (제3정규형)

이행적 함수 종속 제거 (키가 아닌 속성에 종속된 속성 제거)

학생(학번, 이름, 학과코드, 학과이름) → 학과이름은 학과코드에 종속 → 별도 학과 테이블로 분리

◆ 그 외 정규형 (고급 단계)

정규형	설명
BCNF (보이스-코드 정규형)	모든 결정자가 후보키
4NF	다치 종속 제거 (예: 하나의 키에 여러 종속값이 있을 때)
5NF	조인 종속성 제거

✅ 정규화 vs 비정규화

구분	정규화	비정규화
목적	중복 제거, 무결성 확보	성능 향상, JOIN 최소화
장점	일관성, 데이터 구조 명확	읽기 성능 향상
단점	JOIN이 많아져 성능 저하 가능	데이터 중복, 무결성 위험

📌 실무에선 읽기 성능이 중요한 경우 비정규화를 부분적으로 허용하기도 함

정규화를 하지 않을 경우, 발생할 수 있는 이상현상에 대해 설명해 주세요.

정규화를 하지 않으면 생기는 **3가지 이상현상 (Anomalies)**

정규화를 하지 않고 테이블에 중복되고 종속적인 데이터를 함께 저장하면, 삽입(Insert), 삭제(Delete), 갱신(Update) 시 데이터 불일치 또는 손실이 발생할 수 있습니다.

1 삽입 이상 (Insertion Anomaly)

데이터를 추가하고 싶은데, 불필요한 다른 정보도 함께 넣어야만 하는 경우

📌 예시

학번	이름	과목명	교수명
1	철수	DB	김교수
2	영희	NULL	NULL

→ 영희는 아직 수강 과목이 없지만 학생 등록을 하려면 과목과 교수도 함께 입력해야 함

✅ 정규화를 하면 학생 정보와 수강 정보는 분리된 테이블로 관리되어 문제 없음

2 삭제 이상 (Deletion Anomaly)

어떤 데이터를 삭제했더니, 관련된 다른 정보도 함께 사라지는 문제

📌 예시

학번	이름	과목명	교수명
1	철수	DB	김교수

→ 철수 학생을 삭제하면 김교수의 강의 정보까지 사라짐

✅ 정규화를 하면 교수 정보는 별도 테이블에 있어 안전

3 갱신 이상 (Update Anomaly)

중복된 데이터를 수정할 때 하나만 수정하면 데이터가 불일치하게 되는 문제

📌 예시

학번	이름	과목명	교수명
1	철수	DB	김교수
2	영희	DB	김교수

→ 김교수가 퇴사해서 이름을 “이전교수”로 바꾸려면

→ 모든 레코드를 다 수정해야 함, 하나라도 빠지면 불일치 발생

✅ 정규화를 하면 교수 정보는 하나의 테이블에 1번만 저장됨

✅ 정리: 정규화 미흡 시 발생하는 이상 현상

이상 종류	발생 원인	문제 예시
삽입 이상	한 테이블에 너무 많은 정보를 함께 저장	학생 등록하려면 과목 정보도 필요
삭제 이상	정보 간 종속 관계가 강함	학생 삭제 시 교수 정보까지 사라짐
갱신 이상	중복된 데이터 존재	교수 이름 하나만 바꿨더니 일관성 깨짐

각 정규화에 대해, 그 정규화가 진행되기 전/후의 테이블의 변화에 대해 설명해 주세요.

✅ 가정: 아래는 정규화 전의 하나의 테이블입니다

학생ID	학생이름	과목명	교수이름	교수전화번호
1	철수	DB, 자료구조	김교수	010-1111-1111
2	영희	자료구조	김교수	010-1111-1111
3	민수	운영체제, 컴퓨터구조	이교수	010-2222-2222

📌 여기에는 다음과 같은 문제가 존재합니다:

- 한 셀에 여러 과목 저장 (1NF 위반)
- 교수 정보 중복 (3NF 위반)
- 과목이 학생ID에 부분적으로만 종속 (2NF 위반 가능)

✅ 1. 제1정규형 (1NF) — 원자값으로 분리

한 셀에 여러 값이 있으면 안 된다 → "과목명"에서 복수 과목 제거

◆ 정규화 전 (1NF 위반)

학생ID	학생이름	과목명
1	철수	DB, 자료구조

◆ 정규화 후 (1NF 적용)

학생ID	학생이름	과목명
1	철수	DB
1	철수	자료구조

✅ 이렇게 하면 모든 컬럼이 원자값(atomic value)만 가지게 됨

✅ 2. 제2정규형 (2NF) — 부분 함수 종속 제거

기본키가 복합키일 때, 기본키의 일부에만 종속된 속성은 분리

◆ 정규화 전

학생ID	과목명	학생이름
1	DB	철수
1	자료구조	철수

- 복합키: (학생ID, 과목명)
- 학생이름은 학생ID에만 종속됨 → 부분 종속 발생

◆ 정규화 후 (2NF 적용)

1) 학생 테이블

학생ID	학생이름
1	철수

2) 수강 테이블

학생ID	과목명
1	DB
1	자료구조

✅ 이렇게 하면 부분 종속 제거, 논리적으로 더 정리됨

✅ 3. 제3정규형 (3NF) — 이행적 종속 제거

기본키가 아닌 컬럼이 또 다른 컬럼에 종속되면 분리
→ 예: 교수전화번호는 교수이름에 종속

◆ 정규화 전

과목명	교수이름	교수전화번호
DB	김교수	010-1111-1111
자료구조	김교수	010-1111-1111

→ 교수전화번호는 과목명 → 교수이름 → 교수전화번호로 종속됨 → 이행적 종속

◆ 정규화 후 (3NF 적용)

1) 과목 테이블

과목명	교수이름
DB	김교수

2) 교수 테이블

교수이름	전화번호
김교수	010-1111-1111

✅ 이렇게 하면 이행적 종속 제거, 모든 속성이 기본키에만 직접 종속됨

✅ 정규화 전/후 전체 요약

단계	정규화 전 문제	정규화 후 테이블 분리
1NF	셀에 여러 값 (비원자)	→ 각 셀 하나의 값만 저장
2NF	기본키 일부에만 종속	→ 부분 종속 컬럼 분리
3NF	기본키 아닌 컬럼 간 종속	→ 이행적 종속 분리 (별도 테이블)

💬 면접용 요약 멘트

정규화는 테이블의 구조를 단계별로 개선하는 과정으로,
1NF에서는 셀에 원자값만 저장, **2NF**에서는 기본키의 일부에만 종속된 컬럼을 제거,
3NF에서는 기본키가 아닌 컬럼 간의 종속성(이행적 종속)을 제거합니다.
 각 단계는 데이터의 중복을 줄이고 무결성을 높이며,
 결과적으로 삽입/삭제/갱신 이상 현상을 방지하는 데 목적이 있습니다.

정규화가 무조건 좋은가요? 그렇지 않다면, 어떤 상황에서 역정규화를 하는게 좋은지 설명해 주세요.

정규화의 목적 다시 정리

정규화는 데이터 중복을 제거하고, 무결성을 높이며, 논리적으로 일관된 구조를 만들기 위한 설계 방식입니다.

하지만...

✅ 정규화된 테이블 구조는 **JOIN**이 많아지고, 성능이 떨어질 수 있습니다.

→ 따라서 실무에서는 상황에 따라 역정규화(**Denormalization**)를 선택하는 경우도 많습니다.

역정규화(Denormalization)란?

정규화된 테이블을 다시 통합하거나 중복된 데이터를 저장하는 방식으로,
조회 성능 향상, **JOIN** 최소화, 응답 시간 개선을 목표로 합니다.

✅ 역정규화를 고려해야 할 상황

상황	설명
◆ JOIN 이 너무 많아 성능 저하	복잡한 쿼리로 인해 응답 시간이 늦어질 때
◆ 조회가 매우 빈번하고 빠른 응답이 중요	실시간 서비스, API 응답 속도 최적화
◆ 정적(자주 바뀌지 않는) 데이터가 자주 조회될 때	예: 상품 이름, 카테고리명 등
◆ 보고서, 통계성 데이터	한번에 많은 데이터를 모아서 보여줘야 하는 경우
◆ 데이터 일관성보다는 속도와 효율이 중요한 경우	캐시 저장, 로그 데이터 등

정규화는 데이터의 무결성과 일관성을 보장하기 위한 강력한 설계 방식이지만, 복잡한 **JOIN**으로 인해 성능 저하가 발생할 수 있기 때문에, 실무에서는 조회 성능이나 시스템 요구사항에 따라 역정규화를 선택하는 경우도 많습니다. 특히 조회 성능이 매우 중요한 실시간 서비스나 보고서/통계성 데이터, 또는 정적 데이터가 자주 사용되는 경우에는 중복을 허용하더라도 역정규화를 통해 성능을 최적화하는 것이 효과적일 수 있습니다. 다만, 이로 인해 생길 수 있는 데이터 불일치와 갱신 복잡성을 반드시 고려하여 신중히 결정해야 합니다.

8. View가 무엇이고, 언제 사용할 수 있나요?

View란?

View(뷰)는 하나 이상의 테이블을 기반으로 정의된 가상의 테이블입니다.

실제 데이터를 저장하지 않고, **SELECT** 쿼리 결과를 이름으로 저장한 것처럼 동작합니다.

View의 특징

특징	설명
✓ 가상의 테이블	물리적 데이터 저장 없이 SELECT 결과를 표현
✓ 데이터 재사용	복잡한 쿼리를 재사용 가능 (가독성 ↑)
✓ 보안	민감한 컬럼 제외하고 노출 가능
✓ 독립성 유지	테이블 구조 변경 시에도 View는 그대로 사용 가능 (일부 경우 제외)

View는 언제 사용하나요?

복잡한 **SQL**을 반복해서 사용할 때

- 조인, 서브쿼리, 집계 등 복잡한 쿼리를 뷰로 만들어두면 **재사용이 편리**

```
-- 복잡한 쿼리 생략하고 SELECT * FROM 매출_집계_뷰 WHERE 월 = '2025-03';
```

2 보안 목적

- 예: 직원 테이블에서 급여 정보는 숨기고, 이름과 부서만 보여주고 싶을 때

```
CREATE VIEW public_직원 AS SELECT 이름, 부서명 FROM 직원;
```

3 논리적 독립성 확보

- 실제 테이블 구조가 바뀌더라도 View만 유지되면,
→ View를 사용하는 응용 프로그램은 변경하지 않아도 됨

4 보고서/통계용 데이터 제공

- View를 기반으로 여러 BI 도구나 대시보드에 연동 가능

그렇다면, **View의 값을 수정해도 실제 테이블에는 반영되지 않나요?**

- 단일 테이블에서 생성되고, 기본키를 포함하며 DISTINCT, GROUP BY, 서브쿼리 등을 사용하지 않는 등 복잡한 조건을 만족해야 View를 통해 실제 테이블을 수정할 수 있다.
- 따라서 View를 통한 실제 테이블 변경은 거의 불가능하며, 데이터 정합성을 위해서라도 권장되지 않는다.
- View에서 데이터를 수정하면, 그 View가 **Updatable View**인 경우에만 실제 테이블에 반영됩니다.
수정 가능한 View는 일반적으로 단일 테이블 기반이며, 집계 함수나 조인, 그룹화 없이 단순한 **SELECT**로 만들어진 경우입니다.
반면, 복잡한 View는 기본적으로 읽기 전용이며,
수정이 필요할 경우 **INSTEAD OF** 트리거나 별도 로직으로 처리해야 합니다.

9. DB Join이 무엇인지 설명하고, 각각의 종류에 대해 설명해 주세요.

10. B-Tree와 B+Tree에 대해 설명해 주세요.

11. DB Locking에 대해 설명해 주세요.

12. 트래픽이 높아질 때, DB는 어떻게 관리를 할 수 있을까요?

13. Schema가 무엇인가요?

14. DB의 Connection Pool에 대해 설명해 주세요.

15. Table Full Scan, Index Range Scan에 대해 설명해 주세요.

16. SQL Injection에 대해 설명해 주세요.