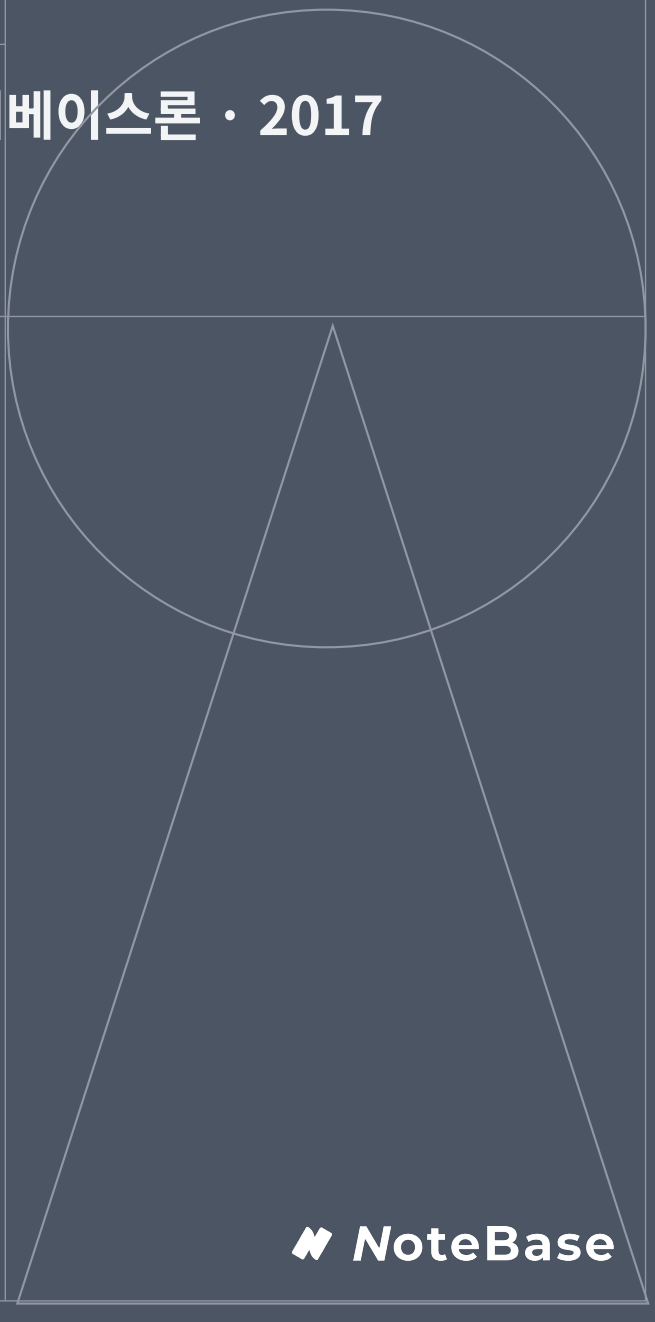
# 기출OX 해설집

공무원/국가직/7급 · 데이터베이스론 · 2017



NoteBase

# 기출OX 해설집

## 공무원/국가직/7급 · 데이터베이스론 · 2017

**공무원/국가직/7급 · 데이터베이스론 · 2017**

**문 1. 데이터베이스 관리 시스템(DBMS)의 역할에 대한 설명으로 옳지 않은 것은?**

① 데이터 조작어(DML)로 스키마의 구조를 기술하여 시스템 카탈로그(혹은 데이터 사전)에 저장한 후 필요할 때 활용한다.

② 질의어 처리기는 질의문을 파싱하고 분석해서 효율적인 데이터베이스 접근코드를 생성한다.

③ 트랜잭션 관리자는 무결성 제약조건 검사, 사용자의 권한 검사, 병행제어, 회복 등의 작업을 수행한다.

④ 저장 데이터 관리자는 디스크에 저장되어 있는 사용자 데이터베이스와 시스템 카탈로그의 접근을 책임진다.

**정답 1번**

**출제 포인트**

DBMS를 구성하는 주요 구성요소(데이터 정의어 컴파일러, 질의어 처리기, 트랜잭션 관리자, 저장 데이터 관리자 등)가 각각 어떤 역할을 담당하는지 정확히 구분할 수 있는지를 묻는 문제이다. 특히 데이터 정의어(DDL)와 데이터 조작어(DML)의 역할을 혼동하지 않는 것이 핵심이다.

**핵심 개념**

데이터 정의어(DDL)는 테이블, 뷰, 인덱스 등 데이터베이스의 스키마(구조)를 정의하는 언어이며, DDL 컴파일러가 이를 분석하여 그 결과를 시스템 카탈로그(데이터 사전)에 저장한다.
데이터 조작어(DML)는 삽입, 삭제, 수정, 검색 등 저장된 데이터 자체를 다루는 언어이며 스키마 구조를 기술하는 데 쓰이지 않는다.
질의어 처리기(질의어 프로세서)는 사용자가 입력한 질의문을 파싱하고 분석하여 효율적으로 실행할 수 있는 저수준의 데이터베이스 접근 코드(실행 계획)로 변환한다.
트랜잭션 관리자는 무결성 제약조건 검사, 사용자 권한 검사, 병행제어(동시성 제어), 장애 발생 시 회복 등 트랜잭션의 정확한 수행을 보장하는 작업을 수행한다.
저장 데이터 관리자는 디스크에 저장된 실제 사용자 데이터베이스와 시스템 카탈로그에 대한 접근을 책임지는 구성요소이다.

NoteBase

### 선지별 해설

① X, 옳지 않다(정답) → 스키마의 구조를 기술하여 시스템 카탈로그에 저장하는 언어는 데이터 조작어(DML)가 아니라 데이터 정의어(DDL)이다. DML은 데이터베이스에 저장된 데이터를 검색·삽입·삭제·수정하는 데 사용되는 언어이며, 스키마 구조 자체를 정의하지 않는다. 틀린 표현 '데이터 조작어(DML)로 스키마의 구조를 기술' → 올바른 표현 '데이터 정의어(DDL)로 스키마의 구조를 기술'.
② O, 옳은 설명이다 → 질의어 처리기는 사용자의 질의문을 파싱하고 분석하여 효율적인 데이터베이스 접근코드(실행 계획)를 생성하는 역할을 수행한다.
③ O, 옳은 설명이다 → 트랜잭션 관리자는 무결성 제약조건 검사, 사용자 권한 검사, 병행제어, 회복 등을 통해 트랜잭션의 일관성 있는 수행을 보장한다.
④ O, 옳은 설명이다 → 저장 데이터 관리자는 디스크에 저장된 사용자 데이터베이스와 시스템 카탈로그의 접근을 책임지는 구성요소이다.

### 오답 포인트

DML과 DDL의 역할을 뒤바꿔 서술하는 함정에 주의한다. '스키마의 구조를 기술하여 카탈로그에 저장한다'는 표현이 나오면 이는 DDL(그리고 이를 처리하는 DDL 컴파일러)의 역할이지 DML의 역할이 아니다. DML은 어디까지나 데이터 값 자체를 다루는 언어라는 점을 구분해야 한다.

### 다시 나오면 이렇게 판단

선지에서 '~어(語)로 스키마를 기술/정의한다'는 문장이 나오면 그 언어가 DDL인지 DML인지부터 확인한다. 스키마·구조·정의라는 단어가 나오면 DDL, 데이터의 삽입·삭제·수정·검색이라는 단어가 나오면 DML로 연결 짓는다. 나머지 선지들은 각 구성요소(질의어 처리기, 트랜잭션 관리자, 저장 데이터 관리자)의 고유 기능 키워드를 그대로 암기해 두면 쉽게 판별할 수 있다.

### 암기 포인트

DDL(데이터 정의어) = 스키마 구조 정의 → 시스템 카탈로그(데이터 사전)에 저장. DML(데이터 조작어) = 데이터 값의 삽입·삭제·수정·검색. 질의어 처리기 = 질의문 파싱·분석 → 접근코드 생성. 트랜잭션 관리자 = 무결성·권한 검사, 병행제어, 회복. 저장 데이터 관리자 = 디스크의 실제 데이터베이스·카탈로그 접근 담당.

NoteBase

**문 2. 관계형 데이터베이스에 대한 설명으로 옳지 않은 것만을 모두 고른 것은?**

ㄱ. 기본키 속성이 복합 속성인 경우 그 속성의 일부 요소 속성에서 널(NULL) 값을 가질 수 있다.
ㄴ. 슈퍼키는 후보키가 되기 위한 필요충분조건이다.
ㄷ. 릴레이션 R이 릴레이션 S를 참조하는 경우 R의 외래키가 S의 기본키가 아닌 후보키 중 하나를 참조해야 한다.
ㄹ. 테이블에 튜플 삽입 시 엔티티 무결성 혹은 키 제약조건, 도메인 제약조건, 참조 무결성 제약조건이 위배될 수 있다.

① ㄱ, ㄴ
② ㄷ, ㄹ
③ ㄱ, ㄴ, ㄷ
④ ㄱ, ㄷ, ㄹ

**정답 3번**

**출제 포인트**

관계형 데이터베이스의 기본 개념인 기본키·슈퍼키·후보키의 관계, 외래키가 참조할 수 있는 대상, 무결성 제약조건의 종류를 정확히 알고 있는지를 묻는다. ㄱ~ㄹ 중 옳지 않은 설명만을 모두 골라야 하므로 각 진술을 개별적으로 참/거짓 판단한 뒤 조합해야 한다.

**핵심 개념**

개체 무결성(entity integrity) 제약조건은 기본키를 구성하는 어떤 속성도 NULL 값을 가질 수 없다는 규칙이다. 기본키가 복합 속성(여러 속성으로 구성된 키)이면 그 구성 속성 전부가 NULL이 될 수 없으며, 일부 구성 요소만 NULL을 허용하는 예외는 없다.
슈퍼키(super key)는 릴레이션의 튜플을 유일하게 식별할 수 있는 속성(또는 속성 집합)이다.
후보키(candidate key)는 슈퍼키 중에서 더 이상 속성을 제거하면 유일성이 깨지는, 즉 최소성(minimality)을 만족하는 슈퍼키이다. 따라서 모든 후보키는 슈퍼키이지만, 모든 슈퍼키가 후보키인 것은 아니다. 슈퍼키라는 조건은 후보키가 되기 위한 필요조건일 뿐 충분조건은 아니다.
외래키(foreign key)는 참조하는 릴레이션(피참조 릴레이션)의 기본키 값을 참조하여 두 릴레이션 간의 관계를 표현하는 속성이다. 외래키는 피참조 릴레이션의 기본키를 참조하는 것이 원칙이며, 기본키가 아닌 다른 후보키만을 참조해야 한다는 제한은 존재하지 않는다.
튜플을 테이블에 삽입할 때는 개체 무결성(기본키 NULL 금지), 키 제약조건(중복 값 금지), 도메인 제약조건(속성이 정의된 도메인 범위를 벗어나면 안 됨), 참조 무결성(외래키 값이 피참조 릴레이션에 존재하지 않는 값이면 안 됨) 등 여러 제약조건이 동시에 검사되며, 이 중 어느 하나라도 위배되면 삽입이 거부된다.

NoteBase

### 선지별 해설

ㄱ. 옳지 않다 → 기본키가 복합 속성인 경우에도 그 구성 요소 속성 중 어느 하나라도 NULL 값을 가질 수 없다. 개체 무결성 제약조건은 기본키를 이루는 모든 구성 속성에 예외 없이 적용되므로, '일부 요소 속성에서는 NULL을 가질 수 있다'는 설명은 틀렸다.
ㄴ. 옳지 않다 → 슈퍼키는 후보키가 되기 위한 필요조건일 뿐이며 충분조건은 아니다. 후보키가 되려면 슈퍼키이면서 동시에 최소성(구성 속성을 하나라도 제거하면 더 이상 유일성을 보장하지 못하는 성질)까지 만족해야 한다. 따라서 '필요충분조건'이라는 표현은 틀렸다.
ㄷ. 옳지 않다 → 외래키는 피참조 릴레이션 S의 기본키를 참조하는 것이 원칙이다. 'S의 기본키가 아닌 후보키 중 하나를 참조해야 한다'는 설명은 외래키가 기본키를 참조할 수 없다는 취지로 읽히므로 틀렸다.
ㄹ. 옳다 → 튜플 삽입 시 기본키 값이 NULL이거나 이미 존재하는 값이면 개체 무결성·키 제약조건 위반이 되고, 속성 값이 정의된 도메인 범위를 벗어나면 도메인 제약조건 위반, 외래키 값이 피참조 릴레이션에 존재하지 않는 값이면 참조 무결성 제약조건 위반이 된다. 따라서 이 설명은 옳다.

### 오답 포인트

①(ㄱ, ㄴ)은 옳지 않은 설명 중 ㄷ을 빠뜨린 조합이고, ②(ㄷ, ㄹ)은 옳은 설명인 ㄹ을 옳지 않은 것으로 잘못 포함시키고 ㄱ, ㄴ을 빠뜨린 조합이며, ④(ㄱ, ㄷ, ㄹ)는 옳은 설명인 ㄹ을 옳지 않은 것에 포함시키고 옳지 않은 ㄴ을 빠뜨린 조합이다. 슈퍼키·후보키·외래키에 대한 정의를 하나라도 흐릿하게 기억하면 이런 조합 오류가 생기기 쉽다.

### 다시 나오면 이렇게 판단

먼저 각 지문을 '기본키/슈퍼키/후보키'에 관한 것인지, '외래키의 참조 대상'에 관한 것인지, '무결성 제약조건 종류'에 관한 것인지로 분류한다. 그다음 슈퍼키-후보키 관계는 '후보키는 항상 슈퍼키이지만 역은 성립하지 않는다(필요조건)'는 한 문장으로, 외래키는 '기본키를 참조하는 것이 원칙'이라는 한 문장으로 판단 기준을 고정해 두면 조합형 문제에서 실수를 줄일 수 있다.

### 암기 포인트

복합 기본키는 구성 속성 전부가 NULL 불가(예외 없음). 슈퍼키는 후보키의 필요조건이지 필요충분조건이 아니다(최소성 추가 필요). 외래키는 피참조 릴레이션의 기본키를 참조하는 것이 원칙이다. 튜플 삽입 시 검사되는 제약조건은 개체 무결성·키 제약조건·도메인 제약조건·참조 무결성 네 가지이다.

NoteBase

**문 3. 테이블의 튜플 데이터를 파일 내에 저장하기 위한 구조는 힙(heap) 파일구조, 순차(sequential) 파일구조, 해시(hash) 파일구조로 구분될 수 있다. 일반적으로 테이블에 가장 빈번하게 이루어지는 연산에 의해 사용될 파일 구조가 결정된다. 다음에 주어진 각각의 연산에 대하여 가장 효율적인 파일구조를 바르게 연결한 것은?**

ㄱ. 속성 값의 범위 구간을 주고 이에 해당하는 튜플 검색 (range search)
ㄴ. 빈번한 튜플 삽입
ㄷ. 특정 속성 값과 일치하는 튜플 검색

① ㉠: 순차, ㉡: 해시, ㉢: 힙
② ㉠: 순차, ㉡: 힙, ㉢: 해시
③ ㉠: 힙, ㉡: 순차, ㉢: 해시
④ ㉠: 해시, ㉡: 힙, ㉢: 순차

**정답 2번**

**출제 포인트**

힙(heap) 파일구조, 순차(sequential) 파일구조, 해시(hash) 파일구조가 각각 어떤 연산(범위 검색, 삽입, 특정 값 검색)에 효율적인지를 구분할 수 있는지를 묻는 문제이다. 파일구조의 물리적 저장 방식과 그에 따른 연산별 성능 차이를 정확히 짝지어야 한다.

**핵심 개념**

힙 파일구조는 튜플을 정렬 없이 삽입 순서대로 파일 끝에 append하는 방식이다. 삽입 시 정렬 위치를 찾을 필요가 없어 삽입 연산이 가장 빠르지만, 검색은 처음부터 끝까지 순차적으로 훑어야 하므로 비효율적이다.
순차 파일구조는 특정 속성(주로 키 값) 기준으로 튜플을 정렬하여 저장하는 방식이다. 정렬되어 있으므로 범위 구간을 지정한 검색(range search)에서 해당 구간의 시작점을 찾은 뒤 정렬 순서대로 이어서 읽으면 되어 효율적이다. 다만 새 튜플을 삽입할 때는 정렬 순서를 유지하기 위해 삽입 위치를 찾고 뒤 레코드를 밀어내야 하므로 삽입 비용이 크다.
해시 파일구조는 속성 값에 해시 함수를 적용해 저장 위치(버킷)를 계산하는 방식이다. 특정 값 하나와 정확히 일치하는 튜플을 찾는 완전일치 검색(exact match)에서는 해시 함수로 위치를 바로 계산해 접근하므로 매우 효율적이다. 반면 해시값은 순서를 보존하지 않으므로 범위 검색에는 적합하지 않다.

NoteBase

### 선지별 해설

① ㄱ: 순차, ㄴ: 해시, ㄷ: 힙 → X. ㄴ(빈번한 삽입)에는 해시가 아니라 힙이 효율적이고, ㄷ(특정 값 일치 검색)에는 힙이 아니라 해시가 효율적이다. ㄴ과 ㄷ이 서로 바뀌었다.
② ㄱ: 순차, ㄴ: 힙, ㄷ: 해시 → O. ㄱ(범위 검색)은 정렬된 순차 파일구조가 적합하고, ㄴ(빈번한 삽입)은 정렬을 유지할 필요 없이 끝에 추가만 하면 되는 힙 파일구조가 적합하며, ㄷ(특정 값 일치 검색)은 해시 함수로 즉시 위치를 계산하는 해시 파일구조가 적합하다.
③ ㄱ: 힙, ㄴ: 순차, ㄷ: 해시 → X. ㄱ(범위 검색)에는 힙이 아니라 순차가 효율적이고, ㄴ(빈번한 삽입)에는 순차가 아니라 힙이 효율적이다. ㄱ과 ㄴ이 서로 바뀌었다.
④ ㄱ: 해시, ㄴ: 힙, ㄷ: 순차 → X. ㄴ(빈번한 삽입)은 힙으로 맞으나, ㄱ(범위 검색)에는 해시가 아니라 순차가 효율적이고 ㄷ(특정 값 일치 검색)에는 순차가 아니라 해시가 효율적이다. ㄱ과 ㄷ이 서로 바뀌었다.

### 오답 포인트

이 문제의 오답은 모두 '순차-범위검색', '힙-삽입', '해시-완전일치검색'이라는 세 대응 쌍 중 두 개를 서로 맞바꾸는 형태로 구성되어 있다. 힙은 삽입이 빠른 대신 어떤 검색에도 유리하지 않다는 점, 순차는 정렬되어 있어 범위 검색에 강하지만 삽입이 느리다는 점, 해시는 정렬을 포기하는 대신 완전일치 검색에 특화되어 있다는 점을 각각 짝으로 기억해야 헷갈리지 않는다.

### 다시 나오면 이렇게 판단

문제에서 요구하는 연산이 '범위(구간) 검색'인지, '삽입 위주'인지, '특정 값과의 완전일치 검색'인지부터 구분한다. 범위 검색이면 정렬이 되어 있어야 하므로 순차, 삽입 위주이면 정렬 유지 비용이 없는 힙, 완전일치 검색이면 계산으로 바로 위치를 찾는 해시를 선택한다.

### 암기 포인트

순차=범위 검색(정렬 유지), 힙=빈번한 삽입(정렬 불필요, append), 해시=완전일치 검색(해시 함수로 즉시 위치 계산).

NoteBase

**문 4. 부서와 사원 테이블을 생성하는 SQL 문장을 수행한 후 튜플 삽입으로 두 테이블의 상태가 다음과 같을 때, 테이블 연산 수행에 대한 설명으로 옳지 않은 것은?**

```
CREATE TABLE 부서 (
  부서번호    INT   NOT NULL,
  부서명      VARCHAR(20),
  PRIMARY KEY(부서번호));

CREATE TABLE 사원 (
  사번        INT   NOT NULL,
  이름        VARCHAR(20),
  부서번호    INT,
  PRIMARY KEY(사번),
  FOREIGN KEY(부서번호) REFERENCES 부서(부서번호));
```

부서

| 부서번호 | 부서명 |
|---|---|
| 1 | 자재부 |
| 2 | 영업부 |

사원

| 사번 | 이름 | 부서번호 |
|---|---|---|
| 11 | 홍길동 | 1 |
| 12 | 이순신 | 2 |

① 부서 테이블에서 (2, '영업부') 튜플을 삭제한다면 참조 무결성 제약조건을 위배한다.

② 사원 테이블에 (13, '강감찬', 'A1') 튜플을 삽입한다면 도메인 무결성 제약조건을 위배한다.

③ 사원 테이블에 (14, '김유신', 0) 튜플을 삽입한다면 참조 무결성 제약조건을 위배한다.

④ 부서 테이블에 (1, '연구부') 튜플을 삽입한다면 참조 무결성 제약조건을 위배한다.

**정답 4번**

### 출제 포인트

관계형 데이터베이스의 세 가지 무결성 제약조건인 개체 무결성(entity integrity), 참조 무결성(referential integrity), 도메인 무결성(domain integrity)을 구체적인 SQL 상황에 적용해 어떤 위반이 어떤 제약조건에 해당하는지 구분하는 문항이다. 부서 테이블은 부서번호를 기본키로, 사원 테이블은 사번을 기본키로 하고 부서번호를 부서 테이블을 참조하는 외래키로 선언하고 있다.

NoteBase

### 핵심 개념

개체 무결성=기본키는 널이 될 수 없고 서로 다른 튜플끼리 값이 중복될 수 없다는 제약이다. 이미 존재하는 기본키 값과 같은 값을 다시 삽입하려 하면 개체 무결성(기본키 유일성) 위반이다.
참조 무결성=외래키 값은 반드시 참조 대상 테이블에 존재하는 기본키 값이거나 널이어야 한다는 제약이다. 참조되고 있는 기본키 행을 삭제하거나, 참조 테이블에 존재하지 않는 값을 외래키에 삽입하면 참조 무결성 위반이다.
도메인 무결성=속성 값이 그 속성에 정의된 데이터 타입과 범위(도메인)를 벗어나지 않아야 한다는 제약이다. INT로 선언된 속성에 문자열을 넣으면 도메인 무결성 위반이다.

### 선지별 해설

① 부서 테이블에서 (2, '영업부') 튜플을 삭제한다면 참조 무결성 제약조건을 위배한다 → O. 사원 테이블의 이순신 튜플이 부서번호=2를 외래키로 참조하고 있으므로, 부서번호 2인 부서 행을 삭제하면 참조하는 외래키 값이 부서 테이블에 존재하지 않게 되어 참조 무결성을 위배한다.
② 사원 테이블에 (13, '강감찬', 'A1') 튜플을 삽입한다면 도메인 무결성 제약조건을 위배한다 → O. 사원 테이블의 부서번호 속성은 INT로 선언되어 있는데 'A1'은 문자열이므로 해당 속성의 도메인(정수형)을 벗어나 도메인 무결성을 위배한다.
③ 사원 테이블에 (14, '김유신', 0) 튜플을 삽입한다면 참조 무결성 제약조건을 위배한다 → O. 부서번호 0은 부서 테이블에 존재하지 않는 값인데, 외래키는 널이거나 참조 테이블에 존재하는 값이어야 하므로 참조 무결성을 위배한다.
④ 부서 테이블에 (1, '연구부') 튜플을 삽입한다면 참조 무결성 제약조건을 위배한다 → X. 부서번호 1은 이미 부서 테이블에 (1, '자재부')로 존재하는 기본키 값이므로, 이를 다시 삽입하면 기본키 중복이 되어 개체 무결성 제약조건을 위배하는 것이다. 참조 무결성은 외래키가 참조하는 값의 존재 여부에 관한 제약이므로 부서 테이블에 대한 삽입 자체와는 관련이 없다.

### 오답 포인트

④를 참조 무결성 위반으로 착각하기 쉬운 이유는 부서 테이블이 사원 테이블로부터 참조당하는 대상 테이블이기 때문이다. 그러나 참조 무결성은 외래키를 가진 테이블(사원)이 참조 대상(부서)에 없는 값을 가리키거나, 참조 대상의 값이 삭제·변경되어 외래키가 고아가 되는 상황에서 문제되는 것이지, 참조 대상 테이블(부서)에 이미 있는 기본키 값을 중복 삽입하는 것은 그 테이블 자신의 기본키 유일성, 즉 개체 무결성 문제다. 삭제(①)와 삽입(④)을 같은 참조 무결성으로 뭉뚱그리지 않도록 주의해야 한다.

### 다시 나오면 이렇게 판단

삽입·삭제 대상이 어느 테이블이고 그 테이블에서 문제되는 속성이 기본키인지 외래키인지를 먼저 구분한다. 기본키 값이 중복되면 개체 무결성, 외래키 값이 참조 테이블에 없거나 참조되던 기본키가 삭제·변경되면 참조 무결성, 속성 값의 타입·범위가 선언과 다르면 도메인 무결성으로 분류한다.

NoteBase

**암기 포인트**

개체 무결성=기본키 중복·널 금지. 참조 무결성=외래키는 참조 테이블의 기본키 값이거나 널. 도메인 무결성=속성 값은 선언된 타입·범위 준수. 부서(참조 대상) 테이블에 기존 기본키 값을 다시 넣으면 개체 무결성 위반, 사원(참조하는 쪽) 테이블에 없는 부서번호를 넣거나 부서 테이블의 참조되는 행을 지우면 참조 무결성 위반이다.

NoteBase

**문 5. 인덱싱(indexing) 기법에 대한 설명으로 옳은 것은?**

① 순서(ordered) 인덱싱은 키 값을 이용하여 찾고자 하는 튜플이 저장된 물리적 주소를 계산해내는 기법이다.

② 차수가 p인 B트리의 모든 노드는 최소한 $\lceil \frac{p}{2} \rceil$개의 트리 포인터를 갖는다.

③ B트리와 B+트리의 노드 크기가 동일한 경우, B+트리의 내부 노드 차수(degree)가 B트리 노드 차수보다 더 크다.

④ 100명이 저장된 학생 테이블의 '성별' 속성에 대하여 비트맵 인덱스를 만들 때, 남학생의 비트맵 인덱스 크기는 남학생 수에 영향을 받는다.

**정답 3번**

**출제 포인트**

인덱싱 기법의 종류(순서 인덱싱, 해시 인덱싱)와 B트리·B+트리·비트맵 인덱스의 구조적 특성을 정확히 구분할 수 있는지를 묻는 문항이다. 특히 B트리와 B+트리는 내부 노드가 저장하는 정보의 차이 때문에 같은 노드 크기에서도 차수가 달라진다는 점이 핵심이다.

**핵심 개념**

순서 인덱싱은 정렬된 파일에 대해 키 값과 그 키가 저장된 레코드의 주소를 담은 인덱스 테이블을 만들고, 이 테이블을 탐색해 주소를 찾아가는 기법이다. 반면 키 값 자체에 해시 함수를 적용해 저장 주소를 직접 계산해내는 방식은 해시(hashing) 기법이다.

B트리는 키 값과 레코드 포인터를 리프 노드뿐 아니라 내부 노드에도 함께 저장한다. B+트리는 내부 노드에는 키 값과 트리 포인터만 저장하고, 실제 레코드를 가리키는 데이터 포인터는 리프 노드에만 저장한다.

차수가 p인 B트리에서 루트 노드를 제외한 모든 노드는 최소 ⌧p/2⌧개의 트리 포인터를 가져야 하지만, 루트 노드는 이 제약에서 예외로 최소 2개의 자식만 있어도 된다.

비트맵 인덱스는 하나의 속성 값(예: 성별의 '남')마다 전체 테이블의 튜플 수만큼 비트를 갖는 비트 벡터를 만들어, 해당 값을 갖는 튜플 위치에는 1을, 그렇지 않으면 0을 부여하는 방식이다.

NoteBase

### 선지별 해설

① X - 키 값을 이용해 저장 주소를 직접 계산해내는 것은 해시 인덱싱의 방식이다. 순서 인덱싱은 정렬된 인덱스 항목을 탐색하여 레코드 주소를 찾는 방식이므로, 주소를 '계산'한다는 설명은 순서 인덱싱이 아니라 해시 인덱싱에 해당한다.
② X - 차수가 p인 B트리에서 최소 ⌈p/2⌉개의 트리 포인터를 가져야 하는 것은 루트 노드를 제외한 나머지 노드에 적용되는 규칙이다. 루트 노드는 최소 2개의 자식만 있어도 되므로, '모든 노드'라는 표현은 틀린 설명이다.
③ O - B트리의 내부 노드는 키 값과 함께 레코드를 가리키는 데이터 포인터도 저장하지만, B+트리의 내부 노드는 데이터 포인터 없이 키 값과 트리 포인터만 저장한다. 따라서 노드 크기가 같다면 B+트리의 내부 노드는 데이터 포인터가 차지할 공간만큼 더 많은 키와 트리 포인터를 담을 수 있어 차수가 B트리보다 더 크다.
④ X - 비트맵 인덱스에서 하나의 값에 대한 비트 벡터의 길이는 그 값을 가진 튜플의 개수가 아니라 테이블 전체의 튜플 수(100명)에 의해 결정된다. 따라서 남학생의 비트맵 인덱스 크기는 남학생 수가 아니라 전체 학생 수인 100에 의해 정해지므로 틀린 설명이다.

### 오답 포인트

순서 인덱싱과 해시 인덱싱의 정의를 뒤바꿔 서술하는 함정에 주의해야 한다. '주소를 계산해낸다'는 표현이 나오면 해시 기법을 의심해야 한다. 또한 B트리의 최소 포인터 수 규칙에서 루트 노드가 예외라는 점을 놓치면 ②를 정답으로 잘못 고르기 쉽다. 비트맵 인덱스의 크기는 값의 빈도가 아니라 전체 튜플 수에 좌우된다는 점도 자주 헷갈리는 부분이다.

### 다시 나오면 이렇게 판단

'주소를 계산' 표현이 나오면 해시 인덱싱, '정렬된 인덱스 테이블 탐색' 표현이 나오면 순서 인덱싱으로 구분한다. B트리 관련 선지에서는 '모든 노드'라는 표현이 있으면 루트 노드 예외 규정을 떠올려 확인한다. B트리와 B+트리를 비교하는 선지가 나오면 내부 노드에 데이터 포인터가 있는지 여부부터 확인해 차수 차이를 판단한다. 비트맵 인덱스의 크기를 묻는 선지가 나오면 특정 값의 개수가 아니라 테이블 전체 튜플 수를 기준으로 판단한다.

### 암기 포인트

해시 인덱싱=키로 주소 계산, 순서 인덱싱=정렬된 인덱스 탐색. B트리 최소 포인터 ⌈p/2⌉개는 루트 제외 규정. B트리는 내부 노드에도 데이터 포인터 저장, B+트리는 내부 노드에 키·트리 포인터만 저장하여 같은 노드 크기면 B+트리 차수가 더 큼. 비트맵 인덱스 크기는 전체 튜플 수 기준.

NoteBase

**문 6. 두 트랜잭션 $T_1$과 $T_2$가 다음과 같은 트랜잭션 스케줄로 실행될 때 발생하는 문제는? (단, 데이터 항목 X와 Y의 초기값은 각각 200과 300이고, read(X)와 write(X)는 각각 트랜잭션이 데이터 항목 X를 읽고 쓰는 연산이다)**

| $T_1$ | $T_2$ |
|---|---|
| read(X);<br>X = X - 20; | |
| | read(X);<br>X = X + 50; |
| write(X);<br>read(Y); | |
| | write(X); |
| Y = Y - 20;<br>write(Y); | |
| | read(Y);<br>Y = Y + 30;<br>write(Y); |

시간 ↓

① 갱신 손실 문제(lost update problem)

② 오손 읽기 문제(dirty read problem)

③ 부정확한 요약 문제(incorrect summary problem)

④ 반복할 수 없는 읽기 문제(unrepeatable read problem)

**정답 1번**

**출제 포인트**

두 트랜잭션이 같은 데이터 항목 X를 동시에 갱신할 때 한쪽의 갱신 결과가 사라지는 갱신 손실(lost update) 상황을 스케줄을 직접 따라가며 판별하는 문제이다. 오손 읽기·부정확한 요약·반복 불가능한 읽기와 구분해서 정확히 짚어내는 것이 핵심이다.

**조건 정리**

X의 초깃값은 200, Y의 초깃값은 300이다. T1은 X에서 20을 빼고 Y에서 20을 빼서 각각 다시 쓴다. T2는 X에 50을 더하고 Y에 30을 더해서 각각 다시 쓴다. 표에 제시된 순서(위에서 아래로, 같은 행은 왼쪽 T1이 먼저)대로 연산이 실행된다.

NoteBase

**사용할 공식**

각 트랜잭션이 자신이 읽은 시점의 값을 기준으로 지역 변수를 계산한 뒤 write 시점에 그 값을 데이터베이스에 반영한다는 원칙만 있으면 된다. 즉 write(X)가 실행되는 순간 X에는 그 write를 실행한 트랜잭션이 read 시점에 읽어 계산해 둔 값이 그대로 저장되며, 뒤에 다른 트랜잭션이 write(X)를 실행하면 앞의 값을 덮어쓴다.

**계산 과정**

1행: T1이 read(X)로 X=200을 읽고 지역적으로 X=200-20=180을 계산해 둔다(아직 DB에 반영 전).
2행: T2가 read(X)로 X=200(T1의 write가 아직 일어나지 않았으므로 원래 값)을 읽고 지역적으로 X=200+50=250을 계산해 둔다.
3행: T1이 write(X)를 실행해 DB의 X를 180으로 갱신한다. 이어서 T1이 read(Y)로 Y=300을 읽는다.
4행: T2가 write(X)를 실행해 DB의 X를 250으로 갱신한다. 이 순간 3행에서 T1이 써 넣은 X=180이 T2의 X=250으로 덮어써져 사라진다.
5행: T1이 Y=300-20=280을 계산해 write(Y)로 DB의 Y를 280으로 갱신한다.
6행: T2가 read(Y)로 Y=280을 읽고 Y=280+30=310을 계산해 write(Y)로 DB의 Y를 310으로 갱신한다.
결과적으로 X는 최종적으로 250이 되어, T1이 수행한 X=X-20 갱신(180)이 완전히 사라지고 반영되지 않았다.

**선지 판단**

① 갱신 손실 문제(lost update problem): O. T1이 write(X)로 반영한 X=180이 뒤이은 T2의 write(X)=250에 의해 덮어써져 사라졌으므로 정확히 갱신 손실 문제에 해당한다.
② 오손 읽기 문제(dirty read problem): X. 오손 읽기는 한 트랜잭션이 아직 커밋되지 않은(롤백될 수도 있는) 다른 트랜잭션의 갱신 값을 읽는 상황인데, 이 스케줄에는 커밋 전 값을 읽어 사용하는 read가 나타나지 않는다. T2가 X를 읽는 시점(2행)은 T1의 write 이전이므로 오손 읽기와 무관하다.
③ 부정확한 요약 문제(incorrect summary problem): X. 부정확한 요약은 한 트랜잭션이 여러 데이터 항목의 합계 등을 집계하는 도중 다른 트랜잭션이 그 항목들을 갱신해 요약 값이 부정확해지는 상황인데, 이 스케줄에는 집계(합계) 연산이 전혀 나타나지 않는다.
④ 반복할 수 없는 읽기 문제(unrepeatable read problem): X. 이는 한 트랜잭션이 같은 데이터 항목을 두 번 읽는 사이에 다른 트랜잭션이 그 값을 변경해 두 읽기 결과가 달라지는 상황인데, 이 스케줄에서 T1과 T2는 각각 X와 Y를 한 번씩만 읽으므로 해당하지 않는다.

**오답 포인트**

T2가 write(X)를 실행하기 전에 T1이 이미 write(X)로 180을 반영해 두었다는 점을 놓치면 단순히 최종값 X=250만 보고 문제를 오손 읽기나 다른 이상 현상으로 착각하기 쉽다. 핵심은 T1의 write 결과가 T2의 write에 의해 값 자체가 사라졌다는 점이며, 이는 커밋 전 값을 읽었기 때문이 아니라 두 트랜잭션이 같은 항목에 순서 없이 write를 겹쳐 실행했기 때문이다.

NoteBase

### 다시 나오면 이렇게 판단

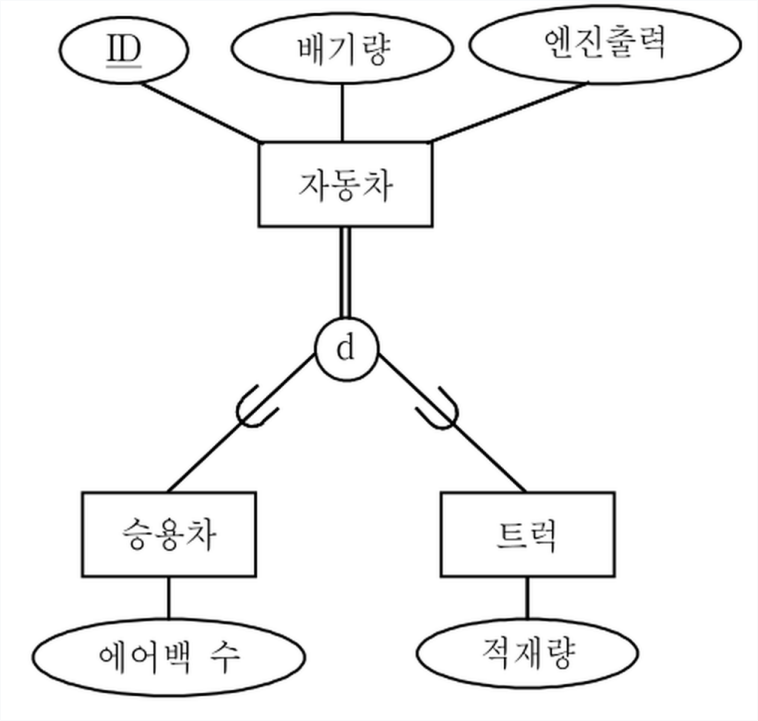
스케줄 문제가 나오면 먼저 각 트랜잭션이 어떤 데이터 항목을 read/write하는지 표로 순서대로 따라가면서, write가 일어난 직후 그 값이 뒤이은 다른 트랜잭션의 write에 의해 그대로 덮어써지는지부터 확인한다. 값이 덮어써져 사라지면 갱신 손실, 커밋 전 값을 읽었으면 오손 읽기, 같은 항목을 두 번 읽었는데 그 사이 값이 바뀌었으면 반복 불가능한 읽기, 집계 도중 값이 바뀌었으면 부정확한 요약으로 구분한다.

### 암기 포인트

갱신 손실=한 트랜잭션의 write 결과가 다른 트랜잭션의 write로 덮어써져 사라짐. 오손 읽기=커밋 전 값을 읽음. 반복 불가능한 읽기=같은 항목을 두 번 읽었는데 값이 달라짐. 부정확한 요약=집계 도중 값이 바뀜.

NoteBase

**문 7. ISA 관계를 슈퍼타입(혹은 상위클래스)과 서브타입(혹은 하위클래스)으로 표현한 EER(Enhanced E-R) 다이어그램의 한 예를 다음 그림에서 보여주고 있다. 이 그림에서 슈퍼타입은 전체 세분화(total specialization)되어 있고, 서브타입들은 서로 분리(disjoint)되어 있다. 이 EER 다이어그램에 대한 설명으로 옳지 않은 것은? (단, 밑줄은 기본키를 의미한다)**




① 트럭 엔터티의 속성의 개수는 4개이다.

② 위 EER 다이어그램에서는 승용차도 아니고 트럭도 아닌 자동차(예를 들면 버스)는 존재할 수 없다.

③ 자동차 엔터티를 테이블로 표현하지 않고, ID, 배기량, 엔진출력 속성을 포함한 승용차와 트럭 테이블로 표현할 수 있다.

④ 승용차와 트럭이 중복(overlap)되었다면 자동차 엔터티는 테이블로 변환할 수 없다.

**정답 4번**

**출제 포인트**

EER 다이어그램의 ISA(슈퍼타입-서브타입) 관계에서 전체 세분화(total specialization, 이중선)와 분리(disjoint, 원 안의 'd') 제약이 각각 무엇을 의미하는지, 그리고 이런 제약 조건에 따라 EER 다이어그램을 관계형 테이블로 변환하는 여러 방식(매핑 옵션) 중 어느 것이 실제로 가능한지를 구분하는 문제이다.

NoteBase

### 핵심 개념

전체 참여(total specialization, 이중선)는 슈퍼타입의 모든 인스턴스가 반드시 하나 이상의 서브타입에 속해야 함을 의미한다. 부분 참여(이중선이 아닌 단일선)는 슈퍼타입 인스턴스가 어떤 서브타입에도 속하지 않을 수 있음을 의미한다.
분리(disjoint, d)는 한 슈퍼타입 인스턴스가 여러 서브타입에 동시에 속할 수 없음을 의미하고, 중복(overlap, o)은 여러 서브타입에 동시에 속할 수 있음을 의미한다.
EER을 관계형 테이블로 바꾸는 대표적인 방식은 세 가지이다. 첫째, 슈퍼타입 테이블 하나와 서브타입 테이블들을 모두 만들고 서브타입 테이블은 슈퍼타입 기본키만 외래키 겸 기본키로 갖는 방식이다. 이 방식은 전체 세분화 여부, 분리·중복 여부와 관계없이 항상 사용할 수 있는 가장 일반적인 방식이다. 둘째, 슈퍼타입 테이블 없이 서브타입 테이블마다 슈퍼타입 속성을 모두 포함시켜 서브타입 테이블만 만드는 방식인데, 이 방식은 전체 세분화이면서 분리(overlap이 아님)일 때만 안전하게 쓸 수 있다. 셋째, 서브타입 구분 없이 하나의 테이블에 모든 서브타입 속성과 타입 구분 속성을 함께 넣는 방식이다.

### 선지별 해설

① 트럭 엔터티는 자신의 고유 속성인 적재량과, 슈퍼타입 자동차로부터 물려받는 ID, 배기량, 엔진출력을 합쳐 총 4개의 속성을 갖는다. 옳은 설명이다.
② 그림에서 자동차와 서브타입 원 'd' 사이는 이중선으로 연결되어 있으므로 전체 세분화(total specialization)이다. 즉 모든 자동차 인스턴스는 반드시 승용차이거나 트럭이어야 하며, 둘 중 어디에도 속하지 않는 자동차(예: 버스)는 존재할 수 없다. 옳은 설명이다.
③ 이 그림은 전체 세분화이면서 동시에 분리(disjoint)이다. 이런 경우에는 슈퍼타입 테이블을 따로 두지 않고, ID·배기량·엔진출력 속성을 승용차와 트럭 테이블에 각각 포함시켜 두 개의 테이블만으로 표현하는 방식이 가능하다. 전체 세분화이므로 자동차이면서 어느 서브타입에도 속하지 않는 경우가 없어 정보 손실이 없고, 분리이므로 한 인스턴스가 두 테이블에 중복 저장될 일도 없기 때문이다. 옳은 설명이다.
④ 승용차와 트럭이 중복(overlap)되었다고 가정해도, 슈퍼타입인 자동차를 별도의 테이블로 만들고 승용차·트럭 테이블이 자동차의 기본키를 공유하는 방식(슈퍼타입 테이블 + 서브타입 테이블 방식)은 여전히 사용할 수 있다. 이 방식은 전체 세분화 여부나 분리·중복 여부에 관계없이 항상 적용 가능한 가장 일반적인 변환 방법이기 때문이다. 따라서 '자동차 엔터티는 테이블로 변환할 수 없다'는 설명은 옳지 않다.

### 오답 포인트

③번을 옳지 않다고 고르기 쉬운데, 이는 슈퍼타입 테이블을 생략하고 서브타입 테이블에 슈퍼타입 속성을 넣는 방식이 항상 불가능하다고 오해하기 때문이다. 이 방식은 전체 세분화이면서 분리일 때는 정보 손실이나 중복 없이 적용 가능하다. 반대로 ④번처럼 중복(overlap) 상황에서 슈퍼타입 테이블 자체를 만들 수 없다고 오해하면 안 된다. 슈퍼타입 테이블 + 서브타입 테이블 방식은 중복이든 분리이든 항상 사용 가능한 범용적인 변환 방법이다.

NoteBase

### 다시 나오면 이렇게 판단

EER 매핑 문제는 먼저 그림에서 이중선(전체/부분 참여)과 d 또는 o(분리/중복) 표시를 정확히 읽어낸 뒤, 각 매핑 방식이 그 조합에서 안전한지를 따진다. 슈퍼타입+서브타입 테이블 방식은 항상 가능하고, 서브타입 테이블만 두는 방식은 전체 세분화이면서 분리일 때만 안전하다는 두 원칙만 기억하면 대부분의 선지를 판단할 수 있다.

### 암기 포인트

이중선=전체 참여(total), 단일선=부분 참여(partial). 원 안의 d=분리(disjoint), o=중복(overlap). 슈퍼타입 테이블+서브타입 테이블 방식은 조건과 무관하게 항상 가능하고, 서브타입 테이블만 두는 방식은 '전체 세분화 + 분리'일 때만 안전하다.

NoteBase

**문 8. 데이터베이스 관리 시스템의 캐시 관리방식에 대한 설명으로 옳지 않은 것은?**

① no-steal 방식에서는 회복과정 중 UNDO와 REDO 연산의 수행이 모두 필요하다.

② 갱신된 페이지의 수가 많고 크기가 큰 경우 no-steal 방식에 비해 steal 방식이 필요한 캐시 버퍼 크기를 줄일 수 있다.

③ 완료된(committed) 트랜잭션에서 갱신된 페이지가 다른 트랜잭션들에서도 빈번히 갱신되는 경우, force 방식에 비해 no-force 방식이 디스크로부터 캐시로 그 페이지를 다시 읽는 횟수를 줄일 수 있다.

④ force 방식은 트랜잭션이 완료되기 전에 그 트랜잭션이 갱신한 모든 페이지들을 디스크에 저장하게 한다.

**정답 1번**

**출제 포인트**

데이터베이스 버퍼(캐시) 관리 정책인 steal/no-steal과 force/no-force가 트랜잭션 회복 시 UNDO·REDO 필요 여부 및 버퍼 운영에 미치는 영향을 이해하는지 묻는 문제이다.

**핵심 개념**

steal 방식=트랜잭션이 완료(commit)되기 전에도 그 트랜잭션이 갱신한(아직 커밋되지 않은) 더티 페이지를 버퍼에서 디스크로 내보낼(steal) 수 있는 방식이다. no-steal 방식=트랜잭션이 커밋되기 전까지는 그 트랜잭션이 갱신한 페이지를 디스크로 내보내지 않는 방식이다. force 방식=트랜잭션이 커밋될 때 그 트랜잭션이 갱신한 모든 페이지를 즉시 디스크에 기록하는 방식이다. no-force 방식=커밋 시점에 즉시 디스크에 기록하지 않고 버퍼에 남겨 둘 수 있는 방식이다. 회복 관점에서 no-steal은 커밋 전 갱신 내용이 디스크에 미리 반영되는 일이 없으므로 UNDO가 필요하지 않고, no-force는 커밋된 내용이 디스크에 아직 반영되지 않았을 수 있으므로 REDO가 필요할 수 있다. 반대로 steal은 커밋되지 않은 갱신이 디스크에 먼저 반영될 수 있어 UNDO가 필요하고, force는 커밋 시점에 이미 디스크 반영이 끝나므로 REDO가 필요하지 않다.

NoteBase

### 선지별 해설

① X (옳지 않음, 정답) — no-steal 방식에서는 트랜잭션이 커밋되기 전까지 그 트랜잭션이 갱신한 더티 페이지가 디스크로 내보내지지 않으므로, 커밋되지 않은 갱신 내용이 디스크에 반영되는 일 자체가 없다. 따라서 회복 시 커밋 전 상태로 되돌리는 UNDO 연산이 필요하지 않다. 틀린 표현 → 올바른 표현: "UNDO와 REDO 연산의 수행이 모두 필요하다" → "UNDO 연산은 필요하지 않으며(force 여부에 따라 REDO만 필요할 수 있다)".
② O (옳음) — no-steal 방식은 트랜잭션이 끝날 때까지 갱신한 모든 더티 페이지를 버퍼에 계속 보유해야 하므로, 갱신 페이지의 수가 많고 크기가 클수록 더 큰 버퍼 공간이 필요하다. 반면 steal 방식은 커밋 전이라도 더티 페이지를 디스크로 내보낼 수 있어 버퍼 공간을 재사용할 수 있으므로, 필요한 캐시 버퍼 크기를 줄일 수 있다.
③ O (옳음) — no-force 방식은 커밋 시점에 곧바로 디스크에 기록하지 않고 페이지를 버퍼에 남겨 둘 수 있다. 커밋된 페이지가 다른 트랜잭션들에서도 계속 빈번히 갱신된다면, 그 페이지를 버퍼에 남겨 두고 계속 갱신에 사용함으로써 디스크로부터 다시 읽어 오는 횟수를 줄일 수 있다. force 방식은 커밋 시점마다 디스크에 기록하므로 이런 이점이 없다.
④ O (옳음) — force 방식은 트랜잭션이 커밋을 완료하는 시점에 그 트랜잭션이 갱신한 모든 페이지를 디스크에 저장하는 것을 보장하는 방식이다.

### 오답 포인트

steal/no-steal은 '언제 디스크에 쓸 수 있는가(커밋 전이냐 후냐)'의 문제이고, force/no-force는 '커밋 시점에 반드시 디스크에 써야 하는가'의 문제로 서로 독립적인 축임을 혼동하지 않아야 한다. no-steal은 UNDO 불필요, steal은 UNDO 필요, force는 REDO 불필요, no-force는 REDO 필요라는 대응 관계를 뒤섞어 ①처럼 no-steal에 UNDO를 결부시키는 함정에 유의해야 한다.

### 다시 나오면 이렇게 판단

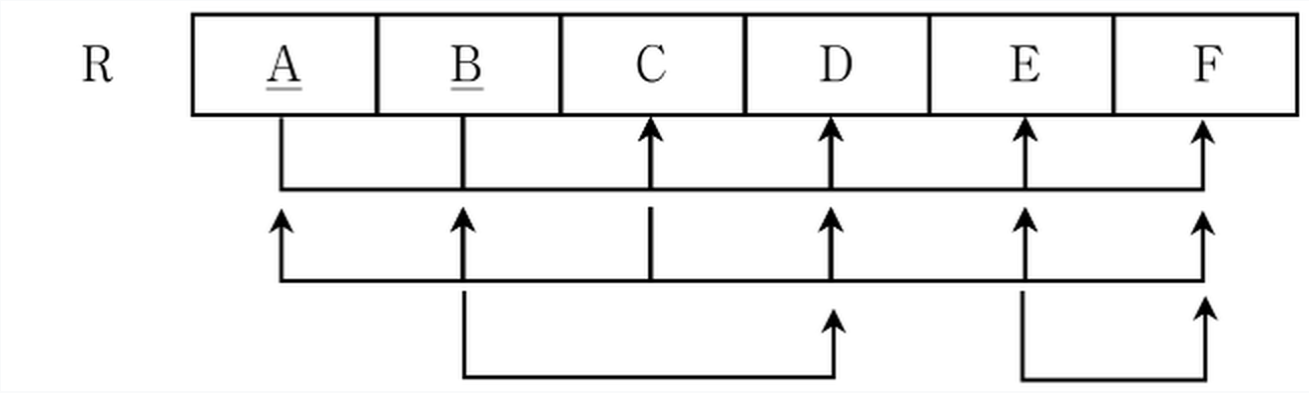
선택지에서 steal·no-steal과 force·no-force라는 단어가 나오면 먼저 두 축을 분리해서 표시해 둔다. no-steal → UNDO 불필요, steal → UNDO 필요, force → REDO 불필요, no-force → REDO 필요로 각각 짝지어 놓고, 선지가 이 대응을 뒤바꿔 서술하는지부터 확인한다.

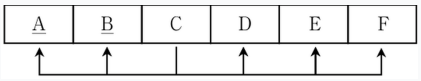
### 암기 포인트

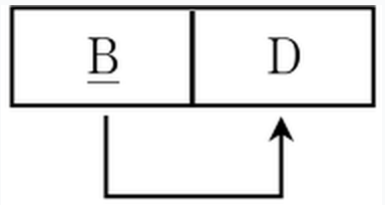
no-steal=UNDO 불필요(커밋 전 디스크 반영 없음), steal=UNDO 필요, force=REDO 불필요(커밋 시 즉시 디스크 반영), no-force=REDO 필요로 짝지어 외워 둔다.

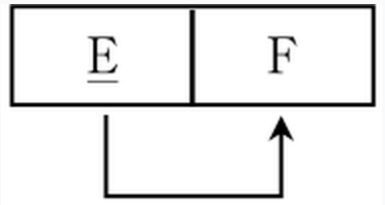
NoteBase

**문 9. 다음 그림은 릴레이션 R과 그 함수적 종속성을 표현하고 있다. 속성 C는 릴레이션 R의 후보키이며, 이 릴레이션은 이미 제1정규화를 수행하였다. 이 릴레이션을 후보키까지 고려하여 제2정규화하였을 때, 분해된 릴레이션 중 기본키가 A, B가 아닌 릴레이션은? (단, 밑줄은 기본키를 의미한다)**




① 


② 


③ 


④ 제2정규화하더라도 변화 없음

**정답 2번**

**출제 포인트**

이 문제는 함수적 종속성(FD) 다이어그램을 읽고 제2정규형(2NF)의 부분함수종속 제거 규칙을 적용하는 문제이다. 릴레이션 R(A, B, C, D, E, F)의 기본키는 복합키 AB이고, 속성 C도 후보키라는 조건이 핵심 단서이다. 후보키가 둘(AB, C)이므로 어떤 속성이 '기본(prime) 속성'인지부터 정확히 가려내야 하고, 그중 복합키 AB의 진부분집합(A만 또는 B만)이 비기본 속성을 결정하는 부분함수종속만 2NF 위반으로 골라내야 한다.

NoteBase

### 자료 읽는 순서

그림의 화살표는 위에서부터 차례로 다음 종속성을 나타낸다. 1) 맨 위 단: A와 B의 선이 합쳐져 C, D, E, F로 화살표가 올라간다 → AB→C,D,E,F(복합키 AB가 나머지 전부를 결정, 기본키이므로 당연). 2) 가운데 단: C에서 내려온 선이 A, B, D, E, F로 화살표가 올라간다 → C→A,B,D,E,F(문제에서 준 대로 C도 후보키이므로 나머지 전부를 결정). 3) 맨 아래 단: B에서 D로 화살표가 올라가는 부분(B→D)과 E에서 F로 화살표가 올라가는 부분(E→F)이 별도로 표시되어 있다. 이 중 AB→CDEF와 C→ABDEF는 각각 후보키 전체가 결정하는 정상적 종속이라 2NF와 무관하다. 문제가 되는 것은 B→D인데, B는 복합키 AB의 진부분집합이고 D는 어느 후보키에도 속하지 않는 비기본 속성이므로, 이는 전형적인 부분함수종속이다. 반면 E→F는 E 자체가 후보키(AB, C)의 부분집합이 아니라 비기본 속성이므로, 이 종속은 부분함수종속이 아니라 이행적 종속(3NF에서 다룰 문제)이며 2NF 분해와는 무관하다.

### 선지별 해설

① 릴레이션 (A,B,C,D,E,F)이 그대로 있고 B→D 화살표까지 그대로 남아 있는 형태 → 틀린 표현: 이는 부분함수종속 B→D를 제거하지 않은 상태이므로 2NF를 만족하지 못한다 → 올바른 표현: D를 분리해 내야 2NF가 완성된다.
② 릴레이션 (B, D), 기본키 B → 옳다. B→D는 복합키 AB의 진부분집합 B가 비기본 속성 D를 결정하는 부분함수종속이므로, 2NF를 만족시키려면 이 종속을 별도 릴레이션 (B, D)로 분리해야 한다. 이 릴레이션의 기본키는 B 하나뿐으로, 원래의 복합키 A,B와는 다른 형태이므로 '기본키가 A, B가 아닌 릴레이션'에 해당한다.
③ 릴레이션 (E, F), 기본키 E → 틀린 표현: E→F는 E가 후보키(AB 또는 C)의 부분집합이 아니므로 부분함수종속이 아니라 이행적 종속에 해당한다 → 올바른 표현: 이행적 종속 제거는 3NF 단계에서 처리할 사항이며, 2NF 분해 결과로 (E,F)가 분리되지는 않는다.
④ 제2정규화하더라도 변화 없음 → 틀린 표현: B→D라는 부분함수종속이 존재하므로 2NF를 만족하지 못해 반드시 분해가 필요하다 → 올바른 표현: (B,D)를 분리해야 2NF가 완성된다.

### 오답 포인트

가장 흔한 실수는 C가 후보키라는 조건을 놓치고 C를 비기본 속성으로 오인해 엉뚱한 종속을 부분함수종속으로 잘못 판단하는 것이다. 두 번째 실수는 E→F를 B→D와 같은 부분함수종속으로 착각하는 것인데, E는 어떤 후보키의 구성 요소도 아니므로 E→F는 애초에 '부분'종속이 성립할 수 없다(부분함수종속은 정의상 결정자가 후보키의 진부분집합이어야 한다). 세 번째 실수는 그림의 화살표 방향(어디가 원인이고 어디가 결과인지)을 반대로 읽어 B→D를 D→B로 잘못 해석하는 것이다.

NoteBase

### 다시 나오면 이렇게 판단

FD 다이어그램 문제가 나오면 먼저 밑줄 친 기본키와 문제 지문에 명시된 다른 후보키를 모두 표시해 기본 속성 집합을 확정한다. 그다음 각 화살표를 '결정자(아래에서 선이 모이는 쪽) → 피결정자(화살표가 향하는 쪽)'로 정리한다. 결정자가 어떤 후보키의 진부분집합이면서 피결정자가 비기본 속성이면 부분함수종속으로 분리 대상이고, 결정자가 비기본 속성이면 그 종속은 이행적 종속이므로 2NF가 아니라 3NF에서 다룬다는 점을 구분한다.

### 암기 포인트

2NF 위반 = 부분함수종속 = '후보키의 일부(진부분집합)'가 '비기본 속성'을 결정하는 경우만 해당한다. 후보키 전체가 결정하는 종속(AB→CDEF, C→ABDEF)이나, 비기본 속성이 다른 비기본 속성을 결정하는 종속(E→F, 이행적 종속)은 2NF와 무관하며 3NF 단계에서 처리한다.

NoteBase

**문 10. 병렬 데이터베이스 구조에 대한 설명으로 옳지 않은 것은?**

① 공유 메모리(shared memory) 구조는 비공유(shared nothing) 구조에 비해 프로세서 간 통신 속도가 빠르지만 프로세서 수의 확장성이 낮다.

② 공유 디스크(shared disk) 구조는 공유 메모리 구조에 비해 메모리 버스의 병목현상과 프로세서 간 통신 속도를 모두 줄인다.

③ 분산 가상 메모리(distributed virtual memory) 구조는 논리적으로는 단일 공유 메모리 구조와 동일하게 사용할 수 있지만 물리적으로는 다중 분리 메모리(multiple disjoint memory) 구조로 구축될 수 있다.

④ 비공유 구조는 공유 디스크 구조에 비해 한 프로세서에서 다른 프로세서가 관리하는 비지역(non-local) 디스크 접근에 대한 비용이 높을 수 있다.

**정답 2번**

**출제 포인트**

병렬 데이터베이스의 대표적 세 가지 구조인 공유 메모리(shared memory), 공유 디스크(shared disk), 비공유(shared nothing) 구조, 그리고 이를 절충한 분산 가상 메모리 구조가 서로 어떤 장단점을 가지는지 정확히 비교하는 문제이다. 특히 각 구조가 프로세서 간 통신 속도와 확장성(스케일러빌리티) 측면에서 어떤 상충 관계에 있는지를 묻는다.

**핵심 개념**

공유 메모리 구조는 모든 프로세서가 하나의 메모리와 디스크를 공유하므로 프로세서 간 통신이 메모리를 통해 이루어져 통신 속도가 가장 빠르지만, 프로세서 수가 늘어날수록 메모리와 버스에 대한 경합(병목현상)이 심해져 확장성이 낮다. 공유 디스크 구조는 각 프로세서가 자신의 로컬 메모리를 따로 가지고 디스크만 공유하므로 메모리 버스의 병목현상은 줄어들지만, 프로세서 간 통신은 더 이상 공유 메모리를 통해 직접 이루어질 수 없고 네트워크를 거쳐야 하므로 통신 속도는 오히려 느려진다. 비공유 구조는 메모리와 디스크를 모두 프로세서별로 분리하여 확장성은 가장 높지만, 다른 프로세서가 관리하는 자원(비지역 디스크 등)에 접근할 때 비용이 커질 수 있다. 분산 가상 메모리 구조는 물리적으로 분리된 여러 메모리를 논리적으로 하나의 공유 메모리처럼 사용할 수 있게 만든 절충 구조이다.

NoteBase

### 선지별 해설

① O — 공유 메모리 구조는 프로세서들이 동일한 메모리를 공유하므로 프로세서 간 통신이 메모리 접근만으로 이루어져 비공유 구조보다 통신 속도가 빠르다. 다만 프로세서 수가 늘어나면 메모리·버스 경합이 커져 확장성은 낮아진다. 옳은 설명이다.
② X — 공유 디스크 구조는 각 프로세서가 자신의 로컬 메모리를 사용하므로 공유 메모리 구조에 비해 메모리 버스의 병목현상은 줄어드는 것이 맞지만, 프로세서 간 통신은 더 이상 메모리를 통해 직접 이루어질 수 없고 별도의 통신망을 거쳐야 하므로 통신 속도는 오히려 줄지 않고 느려진다. 즉 '통신 속도를 줄인다'는 표현이 틀린 표현이며, 올바른 표현은 '통신 속도는 오히려 느려진다(통신 오버헤드가 늘어난다)'이다. 병목현상 감소와 통신 속도 감소를 동시에 얻는다는 서술이 틀렸다.
③ O — 분산 가상 메모리 구조는 물리적으로는 여러 대의 분리된 메모리(다중 분리 메모리)로 구성되어 있지만, 소프트웨어적으로 주소 공간을 통합 관리하여 논리적으로는 하나의 공유 메모리처럼 사용할 수 있게 해 주는 절충형 구조이다. 옳은 설명이다.
④ O — 비공유 구조는 각 프로세서가 자신의 디스크만을 직접 관리하므로, 다른 프로세서가 관리하는 비지역 디스크에 접근하려면 해당 프로세서를 거쳐야 하는 통신 오버헤드가 발생하여 공유 디스크 구조보다 비용이 높아질 수 있다. 옳은 설명이다.

### 오답 포인트

이 문제의 함정은 ②에서 '병목현상 감소'라는 참인 내용과 '통신 속도 감소'라는 거짓 내용을 하나의 문장에 함께 붙여 놓은 데 있다. 공유 디스크 구조가 메모리 버스 병목을 줄인다는 앞부분만 보고 문장 전체를 옳다고 판단하면 안 되며, 프로세서 간 통신은 오히려 느려진다는 뒷부분까지 함께 검토해야 한다. '줄인다'와 '느려진다(늘어난다)'가 반대 방향임을 놓치지 않아야 한다.

### 다시 나오면 이렇게 판단

병렬 DB 구조 비교 문제는 통신 속도와 확장성이 서로 반비례 관계에 있다는 틀로 접근한다. 공유 메모리(통신 빠름, 확장성 낮음) → 공유 디스크(통신 느려짐, 확장성 중간) → 비공유(통신 느림/비용 발생 가능, 확장성 높음) 순으로 확장성이 높아지는 대신 통신 오버헤드가 커진다는 흐름을 기준으로 각 선지의 인과관계 방향이 맞는지를 확인한다.

### 암기 포인트

공유 메모리=통신 빠름·확장성 낮음, 공유 디스크=버스 병목 감소·통신은 느려짐, 비공유=확장성 최고·비지역 자원 접근 비용 발생 가능, 분산 가상 메모리=논리적 단일 공유+물리적 다중 분리 메모리의 절충 구조로 정리해 둔다.

NoteBase

**문 11. RAID에 대한 설명으로 옳지 않은 것은?**

① 장애가 발생했을 때 손실된 데이터를 복구하기 위해 정상 수행 시 여분의 정보를 별도의 저장 공간에 저장해둠으로써 신뢰성을 높일 수 있다.

② Level 0은 동일한 데이터를 여러 디스크에 중복하여 저장함으로써 읽기 성능을 향상시킨다.

③ Level 3은 별도의 한 개의 디스크에 패러티 비트만을 저장한다.

④ Level 5는 모든 디스크가 읽기 작업에 참여한다.

**정답 2번**

**출제 포인트**

RAID(Redundant Array of Independent Disks)의 각 레벨이 데이터를 어떤 방식으로 나누어 저장하는지, 그리고 그 방식이 신뢰성(중복 저장)을 위한 것인지 성능(스트라이핑)을 위한 것인지를 구분할 수 있는지를 묻는다.

**핵심 개념**

RAID 0=중복 정보 없이 데이터를 여러 디스크에 나누어 저장하는 스트라이핑 방식으로, 읽기·쓰기 성능은 향상되지만 디스크 하나만 고장 나도 데이터가 손실되어 신뢰성은 오히려 낮아진다. RAID 1=미러링으로 동일한 데이터를 두 디스크에 그대로 복제해 신뢰성을 높인다. RAID 3=바이트 단위로 데이터를 여러 디스크에 분산 저장하고, 패리티 비트만 저장하는 전용 디스크를 별도로 둔다. RAID 5=패리티 정보를 특정 디스크에 몰아주지 않고 모든 디스크에 분산 저장하여, 읽기 작업 시 모든 디스크가 고르게 참여할 수 있다.

**선지별 해설**

① O — 장애 발생 시 손실된 데이터를 복구하기 위해 평상시에 여분의 정보(패리티나 복제본)를 별도의 저장 공간에 미리 저장해 두는 것은 RAID가 신뢰성을 높이는 일반적인 원리이므로 옳은 설명이다.
② X — Level 0(RAID 0)은 동일한 데이터를 중복하여 저장하지 않는다. 데이터를 단순히 여러 디스크에 나누어(스트라이핑) 저장함으로써 입출력 성능을 높이는 방식이며, 중복 저장이 없어 신뢰성 측면에서는 오히려 취약하다. 따라서 "동일한 데이터를 여러 디스크에 중복하여 저장함으로써 읽기 성능을 향상시킨다"는 설명은 틀렸다 → 올바른 설명은 "중복 저장 없이 데이터를 여러 디스크에 분산(스트라이핑)하여 저장함으로써 성능을 향상시킨다"이다.
③ O — Level 3(RAID 3)은 데이터를 바이트 단위로 여러 디스크에 분산 저장하고, 그로부터 계산된 패리티 비트만을 별도로 지정된 한 개의 전용 디스크에 저장하는 방식이 맞다.
④ O — Level 5(RAID 5)는 패리티 정보를 하나의 전용 디스크에 몰아 저장하지 않고 모든 디스크에 순환적으로 분산시키므로, 읽기 작업에 모든 디스크가 참여할 수 있다는 설명은 옳다.

NoteBase

### 오답 포인트

RAID 0을 미러링(RAID 1)과 혼동하면 ②를 옳은 설명으로 잘못 판단하기 쉽다. RAID 0은 '중복 저장'이 아니라 '분산 저장(스트라이핑)'이 핵심이며, 중복이 없으므로 오히려 신뢰성이 낮아진다는 점을 기억해야 한다. 또한 패리티를 전용 디스크 한 개에 모으는 방식(RAID 3)과 모든 디스크에 분산시키는 방식(RAID 5)을 구분하지 못하면 ③, ④를 헷갈릴 수 있다.

### 다시 나오면 이렇게 판단

선지에 'Level 0'이 등장하면 곧바로 '중복 저장 없음, 스트라이핑, 성능 향상, 신뢰성 낮음'을 떠올린다. 'Level 1'이 나오면 미러링(중복 저장, 신뢰성 향상)을 떠올리고, 'Level 3'은 전용 패리티 디스크 한 개, 'Level 5'는 패리티 분산 저장(모든 디스크가 읽기·쓰기에 참여)으로 구분해 대입하면 옳고 그름을 빠르게 가릴 수 있다.

### 암기 포인트

RAID 0=중복 없음·스트라이핑·성능↑·신뢰성↓. RAID 1=미러링·중복 저장·신뢰성↑. RAID 3=전용 패리티 디스크 1개·바이트 단위 분산. RAID 5=패리티를 모든 디스크에 분산·모든 디스크가 읽기에 참여.

NoteBase

**문 12. 다음 과일 테이블에 대한 SQL 문장 내의 비교조건을 해석한 것으로 옳지 않은 것은? (단, 밑줄은 기본키를 의미한다)**

| 과일 | |
|---|---|
| 과일코드 | 과일명 |
| 10 | 오렌지 |
| 15 | 키위 |
| 19 | 파인애플 |

① “21 NOT IN (SELECT 과일코드 FROM 과일)”은 참이다.

② “19 < ANY (SELECT 과일코드 FROM 과일)”은 거짓이다.

③ “15 < ALL (SELECT 과일코드 FROM 과일)”은 참이다.

④ “19 = ALL (SELECT 과일코드 FROM 과일)”은 거짓이다.

**정답 3번**

**출제 포인트**

이 문제는 SQL의 서브쿼리 비교연산자 중 NOT IN, ANY, ALL의 의미를 정확히 이해하고 있는지를 묻는다. 과일 테이블에는 과일코드 10, 15, 19 세 개의 값만 존재하며, 이 구체적인 값들을 각 비교조건에 대입해 참·거짓을 판정할 수 있어야 한다.

**조건 정리**

과일 테이블의 과일코드 전체 집합은 {10, 15, 19}이다. 각 선지는 이 집합을 대상으로 한 서브쿼리 결과에 특정 값을 NOT IN, ANY, ALL 연산자로 비교한 조건이 참인지 거짓인지 서술하고 있다.

**사용할 공식**

NOT IN: 비교값이 서브쿼리 결과 집합에 하나도 속하지 않으면 참이다. ANY(=SOME): 비교값이 서브쿼리 결과 중 적어도 하나의 값과 비교조건을 만족하면 참이다. 즉 $x < \text{ANY}(S)$는 $x < \max(S)$와 동치이다. ALL: 비교값이 서브쿼리 결과의 모든 값에 대해 비교조건을 만족해야 참이다. 즉 $x < \text{ALL}(S)$는 $x < \min(S)$와 동치이고, $x = \text{ALL}(S)$는 집합의 모든 원소가 $x$와 같아야 참이다.

NoteBase

**계산 과정**

① 21 NOT IN {10, 15, 19}: 21은 집합에 속하지 않으므로 조건은 참이다.
② 19 < ANY {10, 15, 19}: 19보다 큰 값이 집합에 있는지 확인하면 10, 15, 19 어느 것도 19보다 크지 않으므로(즉 $19 < \max\{10, 15, 19\} = 19$는 성립하지 않음) 조건은 거짓이다.
③ 15 < ALL {10, 15, 19}: 집합의 최솟값 10과 비교하면 15 < 10은 성립하지 않으므로(즉 $15 < \min\{10, 15, 19\} = 10$은 거짓) 조건은 거짓이다.
④ 19 = ALL {10, 15, 19}: 19는 10, 15와 같지 않으므로 조건은 거짓이다.

**선지 판단**

① 21 NOT IN (SELECT 과일코드 FROM 과일)은 참이다. → O. 21은 {10, 15, 19}에 속하지 않으므로 실제로 참이며, 선지의 해석도 참이라고 서술해 옳다.
② 19 < ANY (SELECT 과일코드 FROM 과일)은 거짓이다. → O. 19보다 큰 과일코드가 없으므로 실제 조건은 거짓이며, 선지의 해석도 거짓이라고 서술해 옳다.
③ 15 < ALL (SELECT 과일코드 FROM 과일)은 참이다. → X. 15는 최솟값 10보다 작지 않으므로(15 < 10은 성립하지 않음) 실제 조건은 거짓인데, 선지는 참이라고 서술하여 틀렸다. 틀린 표현 "참이다" → 올바른 표현 "거짓이다".
④ 19 = ALL (SELECT 과일코드 FROM 과일)은 거짓이다. → O. 과일코드 값이 모두 19와 같지는 않으므로(10, 15가 존재) 실제 조건은 거짓이며, 선지의 해석도 거짓이라고 서술해 옳다.

**오답 포인트**

가장 흔한 실수는 ALL을 ANY처럼 '적어도 하나'로 착각하는 것이다. $x < \text{ALL}(S)$는 집합의 모든 값보다 작아야 하므로 집합의 최솟값하고만 비교하면 되는데, 이를 최댓값과 비교하거나 '하나라도 작으면 참'이라고 오해하면 ③을 참으로 잘못 판단하게 된다. 반대로 ANY는 최댓값 하나만 만족해도 참이 되는 조건임을 혼동하지 않아야 한다.

**다시 나오면 이렇게 판단**

서브쿼리 비교연산자 문제가 나오면 먼저 서브쿼리 결과 집합을 구체적인 숫자로 나열한 뒤, ANY는 '집합의 최댓값 또는 최솟값 중 유리한 쪽 하나만 만족하면 참', ALL은 '집합의 모든 값에 대해 성립해야 참'이라는 기준으로 하나씩 대입해 판정한다. 부등호 방향에 따라 ANY와 ALL이 비교해야 할 대표값(최댓값/최솟값)이 달라짐에 주의한다.

**암기 포인트**

$x < \text{ANY}(S) \Leftrightarrow x < \max(S)$, $x > \text{ANY}(S) \Leftrightarrow x > \min(S)$, $x < \text{ALL}(S) \Leftrightarrow x < \min(S)$, $x > \text{ALL}(S) \Leftrightarrow x > \max(S)$로 기억해 둔다. NOT IN은 집합에 아예 속하지 않는지를, = ALL은 집합의 모든 값이 특정 값과 같은지를 확인하는 연산자임을 함께 정리해 둔다.

NoteBase

**문 13. 데이터베이스의 위험 요소와 보안에 대한 설명으로 옳지 않은 것은?**

① 데이터베이스에 대한 무결성(integrity) 제약조건이란 부적절한 갱신으로부터 데이터를 보호하기 위한 조건이다.

② 데이터베이스에 대한 접근 제어는 권한이 없는 사용자나 프로그램이 데이터를 검색하거나 변경하는 것을 방지하기 위한 것이다.

③ GRANT와 REVOKE는 데이터베이스에 대한 접근 권한을 부여하거나 취소하기 위한 SQL 명령어이다.

④ 통계 데이터베이스는 원본 데이터로부터 통계 정보를 추출한 것이므로 보안의 대상이 아니다.

**정답 4번**

**출제 포인트**

데이터베이스 보안의 대상과 수단(무결성 제약조건, 접근 제어, 권한 부여·회수 명령어, 통계 데이터베이스 보안)을 구분해서 아는지 묻는 문항이다. 통계 데이터베이스도 추론 공격의 대상이 되어 별도의 보안 대책이 필요하다는 점이 핵심이다.

**핵심 개념**

무결성 제약조건은 부적절한 갱신·삽입·삭제로부터 데이터의 정확성과 일관성을 지키기 위한 규칙이다. 접근 제어는 인가받지 않은 사용자나 프로그램의 검색·변경을 막는 보안 기법이다. GRANT와 REVOKE는 각각 권한을 부여하고 회수하는 SQL 명령어로 접근 제어를 구현하는 수단이다. 통계 데이터베이스는 개별 레코드가 아닌 통계값(합계, 평균, 개수 등)만 제공하는 것처럼 보이지만, 여러 통계 질의 결과를 조합하면 특정 개인의 원본 데이터를 역으로 추론해 낼 수 있는 추론 공격(inference attack)의 위험이 있어 오히려 별도의 보안 통제가 필요한 대상이다.

**선지별 해설**

① O - 무결성 제약조건은 도메인 제약, 개체 무결성, 참조 무결성 등을 통해 부적절한 갱신으로부터 데이터를 보호하는 조건이므로 옳은 설명이다.
② O - 접근 제어는 인가되지 않은 사용자나 프로그램이 데이터를 검색·변경하지 못하도록 통제하는 기능이므로 옳은 설명이다.
③ O - GRANT는 권한 부여, REVOKE는 권한 회수를 수행하는 SQL의 데이터 제어어(DCL) 명령어이므로 옳은 설명이다.
④ X - 틀린 표현: "보안의 대상이 아니다" → 올바른 표현: "통계 데이터베이스도 여러 통계 질의를 조합한 추론 공격으로 개인 정보가 노출될 수 있어 보안 대책(질의 제한, 데이터 은폐, 임의 표본 추출 등)이 필요한 보안의 대상이다."

NoteBase

### 오답 포인트

통계 정보만 제공하니 원본이 감춰져 안전하다고 단순하게 생각하면 ④를 옳은 설명으로 착각하기 쉽다. 그러나 통계값들 사이의 교집합, 차집합을 이용한 추적 질의로 특정 개인의 값을 역산할 수 있기 때문에 통계 데이터베이스는 대표적인 보안 취약 대상 중 하나로 다뤄진다.

### 다시 나오면 이렇게 판단

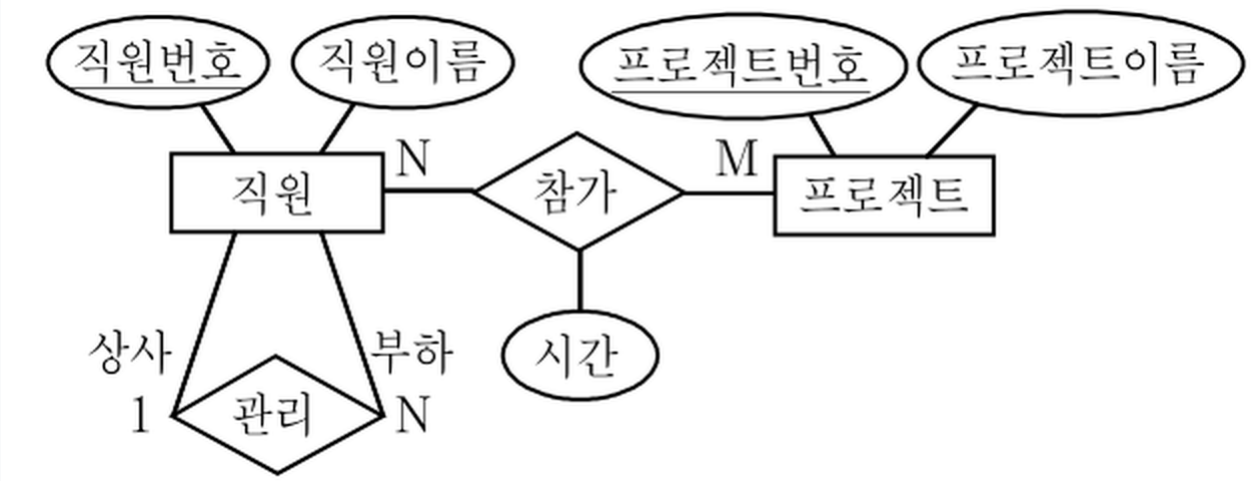
통계 데이터베이스, 추론 공격, 질의 제한 같은 표현이 나오면 우선 '통계 데이터베이스=보안 불필요'라는 서술을 의심한다. 반대로 무결성 제약조건, 접근 제어, GRANT/REVOKE에 대한 정의는 교과서적 정의와 그대로 일치하는지만 확인하면 대부분 옳은 선지로 처리된다.

### 암기 포인트

통계 DB는 개별 데이터를 감추지만 추론 공격에 취약하므로 반드시 보안 대책이 필요하다는 점을 기억해 둔다. 접근 제어의 실행 도구가 GRANT(부여)와 REVOKE(회수)라는 것도 함께 묶어 정리해 둔다.

NoteBase

**문 14. 다음 ERD(Entity-Relationship Diagram)를 관계형 데이터베이스 스키마로 가장 적절하게 변환한 것은? (단, 밑줄은 기본키, 이탤릭체는 외래키를 의미한다)**




① 직원(직원번호, 직원이름), 관리(*상사직원번호*, *부하직원번호*), 프로젝트(프로젝트번호, 프로젝트이름), 참가(시간)

② 직원(직원번호, 직원이름), 프로젝트(프로젝트번호, 프로젝트이름), 참가(*직원번호*, *프로젝트번호*, 시간)

③ 직원(직원번호, 직원이름, *상사직원번호*), 프로젝트(프로젝트번호, 프로젝트이름), 참가(*직원번호*, *프로젝트번호*, 시간)

④ 직원(직원번호, 직원이름, *부하직원번호*), 프로젝트(프로젝트번호, 프로젝트이름), 참가(*직원번호*, *프로젝트번호*, 시간)

**정답 3번**

### 출제 포인트

ERD를 관계형 스키마로 변환할 때 관계(릴레이션십)의 종류에 따라 외래키를 어디에 두는지를 정확히 아는지 묻는 문제이다. 하나는 N:M 관계(참가)이고 다른 하나는 1:N 재귀 관계(관리)인데, 두 관계는 외래키 배치 규칙이 서로 다르다.

### 핵심 개념

1:N 관계는 별도의 관계 릴레이션을 만들지 않고, N쪽(다수 쪽) 개체의 릴레이션에 1쪽 개체의 기본키를 외래키로 추가한다. 관리 관계에서 상사는 1, 부하는 N이므로 부하 역할을 하는 직원 레코드에 상사의 직원번호를 외래키로 추가하면 된다. 이때 상사와 부하가 모두 직원 개체이므로 별도 개체가 아니라 직원 릴레이션 자기 자신에 상사직원번호 속성을 추가하는 형태가 된다. 반면 N:M 관계는 반드시 별도의 관계 릴레이션을 만들어야 하며, 그 릴레이션의 기본키는 양쪽 개체의 기본키를 합친 복합키가 되고 관계 자체의 속성(시간)도 함께 저장한다. 참가 관계는 직원 N : 프로젝트 M이므로 참가(직원번호, 프로젝트번호, 시간) 형태의 별도 릴레이션이 필요하다.

NoteBase

### 선지별 해설

① 직원(직원번호, 직원이름), 관리(상사직원번호, 부하직원번호), 프로젝트(프로젝트번호, 프로젝트이름), 참가(시간) → X. 1:N 관계인 관리를 별도 릴레이션으로 만든 것이 틀렸다(1:N은 별도 릴레이션이 필요 없다). 또한 참가 릴레이션에 직원번호·프로젝트번호가 빠지고 시간만 있어 참가 관계를 식별할 수 없다.
② 직원(직원번호, 직원이름), 프로젝트(프로젝트번호, 프로젝트이름), 참가(직원번호, 프로젝트번호, 시간) → X. 참가 릴레이션은 올바르게 구성했지만, 1:N 관계인 관리(상사-부하)를 표현할 속성이 직원 릴레이션에 전혀 없어 상사 정보가 누락되었다.
③ 직원(직원번호, 직원이름, 상사직원번호), 프로젝트(프로젝트번호, 프로젝트이름), 참가(직원번호, 프로젝트번호, 시간) → O. 1:N 재귀 관계인 관리는 N쪽인 부하 직원 레코드에 상사직원번호를 외래키로 추가해 표현하고, N:M 관계인 참가는 직원번호·프로젝트번호를 복합키로 갖는 별도 릴레이션으로 표현했다. 두 관계 모두 규칙에 맞게 변환되었다.
④ 직원(직원번호, 직원이름, 부하직원번호), 프로젝트(프로젝트번호, 프로젝트이름), 참가(직원번호, 프로젝트번호, 시간) → X. 참가 릴레이션은 맞지만, 관리 관계의 외래키를 부하직원번호로 잘못 두었다. 관리에서 상사가 1쪽이므로 외래키(상사직원번호)는 부하 역할을 하는 직원 레코드 쪽에 상사를 가리키는 형태로 들어가야 하는데, 반대로 부하를 가리키는 속성을 넣으면 한 직원(상사)이 여러 부하를 가질 수 있는 1:N 구조를 하나의 속성값으로 표현할 수 없어 관계가 성립하지 않는다.

### 오답 포인트

1:N 관계와 N:M 관계의 변환 규칙을 혼동하면 틀린다. 1:N 관계(관리)는 별도 릴레이션을 만들 필요 없이 N쪽 릴레이션에 외래키 한 속성만 추가하면 되는데, 이를 별도 릴레이션으로 만들면 ①처럼 오답이 된다. 또한 1:N 관계에서 외래키는 반드시 N쪽(다수 쪽, 여기서는 부하 역할을 하는 직원 레코드)에 1쪽의 기본키를 넣어야 하는데, 방향을 반대로 생각해 N쪽의 기본키를 1쪽에 넣으려 하면(부하직원번호를 직원 릴레이션에 넣는 ④) 한 값으로 여러 부하를 표현할 수 없어 오답이 된다.

### 다시 나오면 이렇게 판단

ERD 변환 문제가 나오면 먼저 관계마다 카디널리티(1:1, 1:N, N:M)를 표시된 숫자(1, N, M 등)로 확인한다. 1:N이면 '별도 릴레이션 없이 N쪽에 1쪽의 기본키를 외래키로 추가'라는 규칙만 적용하고, N:M이면 '반드시 별도 릴레이션을 만들고 그 기본키는 양쪽 기본키의 조합, 관계 속성도 함께 저장'이라는 규칙을 적용한다. 재귀 관계(같은 개체끼리의 관계)도 규칙은 동일하되, 외래키 속성 이름에 역할(상사, 부하 등)을 붙여 구분한다는 점만 유의한다.

### 암기 포인트

1:N은 N쪽에 외래키 한 줄 추가, N:M은 무조건 별도 릴레이션(복합키+관계 속성)으로 변환한다. 재귀 1:N 관계는 '1쪽 역할 이름 + 기본키명'을 N쪽 릴레이션 자신에 외래키로 추가한다(예: 상사직원번호).

NoteBase

**문 15. 메인 메모리 데이터베이스(main memory database) 시스템의 동작에 대한 설명으로 옳지 않은 것은?**

① 메인 메모리 데이터베이스와 디스크 기반 데이터베이스가 동일한 데이터를 저장하고 관리한다면 내부 인덱스 구조도 동일해야 한다.

② 메인 메모리의 데이터를 이용하여 연산을 수행하지만 로깅(logging)은 상대적으로 저속의 비휘발성 저장공간(예를 들면 디스크)에 수행해야 하므로 성능 향상에 방해요소가 될 수 있다.

③ 디스크 기반 데이터베이스보다 메인 메모리 데이터베이스가 트랜잭션 처리 속도 면에서 더 좋은 성능을 보인다.

④ 그룹 완료(group commit) 기법은 트랜잭션이 완료를 요청하더라도 즉시 처리하는 대신, 완료를 기다리는 트랜잭션의 개수가 일정 수 이상이 되거나 일정 시간이 경과한 후 완료 연산을 수행하는 것이다.

**정답 1번**

**출제 포인트**

메인 메모리 데이터베이스(MMDB)가 디스크 기반 데이터베이스와 근본적으로 다르게 설계되는 이유를 묻는 문제이다. 데이터가 항상 메모리에 상주한다는 전제가 인덱스 구조, 로깅 방식, 트랜잭션 처리 성능에 어떤 영향을 주는지 이해하고 있는지를 확인한다.

**핵심 개념**

메인 메모리 데이터베이스는 전체 데이터를 주기억장치(RAM)에 상주시켜 디스크 접근으로 인한 지연을 없애는 방식으로 성능을 높인다. 디스크 기반 데이터베이스는 디스크 I/O 횟수를 줄이는 것이 목표이므로 B-tree 계열처럼 페이지 단위 접근에 최적화된 인덱스를 쓰지만, 메인 메모리 데이터베이스는 디스크 I/O 자체가 문제되지 않으므로 T-tree, 해시 인덱스처럼 메모리에서의 포인터 탐색과 캐시 적중률에 최적화된 구조를 사용한다. 따라서 저장·관리하는 데이터가 같아도 인덱스 구조는 서로 다르게 설계되는 것이 일반적이다. 한편 트랜잭션 처리 결과의 영속성을 보장하려면 로그는 비휘발성 저장장치에 기록해야 하는데, 이 로그 기록(디스크 쓰기)이 메인 메모리 데이터베이스 전체 처리 속도의 병목이 될 수 있다. 이를 완화하는 대표적 기법이 그룹 완료(group commit)로, 완료 요청을 개별적으로 즉시 처리하지 않고 일정 개수 또는 일정 시간 단위로 모아 한 번에 로그를 기록함으로써 디스크 쓰기 횟수를 줄인다.

NoteBase

### 선지별 해설

① 옳지 않다. 메인 메모리 데이터베이스와 디스크 기반 데이터베이스는 동일한 데이터를 저장·관리하더라도 접근 매체와 접근 특성이 다르므로 내부 인덱스 구조를 동일하게 유지할 필요가 없다. 디스크 기반 데이터베이스는 디스크 페이지 입출력 횟수를 최소화하는 B-tree류 인덱스를 주로 쓰는 반면, 메인 메모리 데이터베이스는 메모리 내 포인터 탐색과 캐시 효율을 고려한 T-tree, 해시 인덱스 등을 쓰는 것이 일반적이므로 '동일해야 한다'는 설명은 틀렸다. 틀린 표현 → 올바른 표현: '내부 인덱스 구조도 동일해야 한다' → '접근 매체의 특성이 다르므로 인덱스 구조는 서로 다르게 설계되는 것이 일반적이다'.
② 옳다. 트랜잭션의 완료(commit)를 보장하려면 로그를 비휘발성 저장장치(디스크 등)에 기록해야 하는데, 이 디스크 쓰기는 메모리 연산에 비해 느리므로 메인 메모리 데이터베이스 전체의 성능 향상을 제한하는 요소가 될 수 있다.
③ 옳다. 메인 메모리 데이터베이스는 디스크 입출력 지연이 없어 동일한 트랜잭션을 처리할 때 디스크 기반 데이터베이스보다 처리 속도가 빠르다.
④ 옳다. 그룹 완료 기법은 개별 트랜잭션의 완료 요청을 즉시 처리하지 않고, 대기 중인 트랜잭션의 수가 일정 수 이상이 되거나 일정 시간이 지난 시점에 모아서 한 번에 완료 연산(로그 기록)을 수행하여 디스크 쓰기 횟수를 줄이는 기법이다.

### 오답 포인트

'동일한 데이터를 저장·관리한다'는 조건을 '동일한 내부 구조를 가져야 한다'는 결론으로 착각하기 쉽다. 저장되는 데이터의 내용과 그 데이터를 빠르게 찾기 위한 내부 자료구조는 별개의 문제이며, 접근 매체(메모리 vs 디스크)의 특성 차이가 인덱스 설계를 갈라놓는다는 점을 놓치면 안 된다.

### 다시 나오면 이렇게 판단

메인 메모리 데이터베이스 관련 문제에서 '~해야 한다', '~와 동일해야 한다'처럼 구조적 일치를 강제하는 서술이 나오면 먼저 의심한다. 메인 메모리 데이터베이스의 특징은 (1) 디스크 I/O 지연 제거로 처리 속도 향상, (2) 로그는 여전히 비휘발성 저장장치에 써야 하므로 병목 가능성 존재, (3) 그룹 완료 등으로 그 병목을 완화, (4) 인덱스는 메모리 접근에 최적화된 구조(T-tree, 해시 등)를 별도로 사용한다는 네 가지로 정리해 두면 대부분의 선지를 판단할 수 있다.

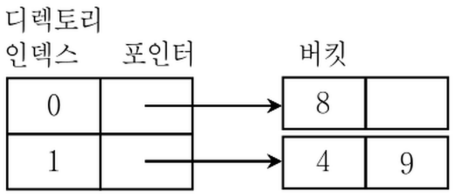
### 암기 포인트

메인 메모리 DB = 데이터는 메모리 상주(빠름) + 로그는 디스크 기록(병목 가능) + 그룹 완료로 로그 병목 완화 + 인덱스는 메모리 특화 구조(디스크 기반 DB와 다름).

NoteBase

**문 16. 다음 그림은 키 값에 대한 해시함수 h의 결과와, 확장 가능 해싱(extendible hashing) 기법에 의해 최초에 키 값 4, 8, 9를 순서대로 입력하여 수행한 결과이다. 이후 키 값 11, 7, 2, 5, 13을 순서대로 입력하는 경우에 대한 설명으로 옳은 것은? (단, 버킷(bucket)의 크기는 2이고 입력된 키 값은 삽입한 순서대로 버킷 내의 왼쪽부터 저장된다)**

| 키 값 | 해시함수 h(키 값) |
|---|---|
| 4 | 1010 |
| 8 | 0001 |
| 9 | 1101 |
| 11 | 1110 |
| 7 | 0110 |
| 2 | 1111 |
| 5 | 0111 |
| 13 | 0101 |




① 키 값 7이 삽입될 때 새 버킷이 생성된다.

② 모든 키 값이 삽입된 후 버킷의 개수는 5이다.

③ 모든 키 값이 삽입된 후 디렉토리 인덱스 010이 가리키는 버킷에는 2개의 키 값이 존재한다.

④ 모든 키 값이 삽입된 후 디렉토리 인덱스 111이 가리키는 버킷의 첫 번째 키 값은 11이다.

**정답 4번**

**출제 포인트**

확장 가능 해싱(extendible hashing)에서 버킷에 오버플로우가 발생했을 때, 분할할 버킷의 지역 깊이(local depth)와 디렉터리의 전역 깊이(global depth)를 비교하여 '디렉터리 전체를 두 배로 늘려야 하는 경우'와 '디렉터리는 그대로 두고 버킷만 분할하면 되는 경우'를 구분해 시뮬레이션할 수 있는지를 확인하는 문제이다. 해시값은 왼쪽(최상위) 비트부터 필요한 자릿수만큼 사용해 디렉터리 인덱스를 결정한다.

NoteBase

### 조건 정리

버킷 크기는 2이고, 같은 버킷에 들어가는 키 값은 삽입한 순서대로 왼쪽부터 채워진다. 초기 상태(전역 깊이 1, 디렉터리 2칸)는 인덱스 0(접두 "0")이 버킷 [8]을, 인덱스 1(접두 "1")이 버킷 [4, 9]를 가리킨다. h(4)=1010, h(8)=0001, h(9)=1101이므로 첫 비트만으로 0/1 배정이 맞아떨어지며, 이후 11(1110), 7(0110), 2(1111), 5(0111), 13(0101)을 이 순서대로 추가 삽입한다.

### 사용할 공식

버킷이 가득 찬 상태에서 새 키가 들어오면 그 버킷을 분할한다. 이때 분할 대상 버킷의 지역 깊이가 디렉터리의 전역 깊이와 같으면 디렉터리 칸 수를 두 배로 늘리고(전역 깊이 +1) 버킷을 둘로 나누어 포인터를 다시 연결한다. 반대로 지역 깊이가 전역 깊이보다 작으면 디렉터리는 늘리지 않고 그 버킷만 지역 깊이를 1 늘려 둘로 분할한 뒤, 분할된 두 버킷을 가리키던 기존 디렉터리 항목들만 재배정한다. 분할된 두 버킷에는 원래 버킷에 있던 키 값들을 새로 늘어난 비트 자리 값(0 또는 1)에 따라 다시 나누어 담는다.

### 계산 과정

11(1110) 삽입: 첫 비트 1이므로 [4, 9] 버킷으로 가야 하는데 이미 가득 차 있어 분할한다. 이 버킷의 지역 깊이(1)가 전역 깊이(1)와 같으므로 디렉터리를 4칸(전역 깊이 2)으로 늘린다. 상위 2비트 기준으로 4(10)는 "10" 버킷에, 9(11)는 "11" 버킷에 들어간다. 11(11)은 "11" 버킷에 들어가 그 버킷은 [9, 11]이 된다.
7(0110) 삽입: 상위 2비트 "01"은 아직 분할되지 않은 [8] 버킷(지역 깊이 1, "0" 접두)으로 연결되어 있으므로 그대로 들어가 [8, 7]이 되고 가득 찬다.
2(1111) 삽입: 상위 2비트 "11" 버킷 [9, 11]이 가득 차 있어 분할한다. 지역 깊이(2)가 전역 깊이(2)와 같으므로 디렉터리를 8칸(전역 깊이 3)으로 늘린다. 상위 3비트로 9(110)는 "110" 버킷에, 11(111)은 "111" 버킷에 들어간다. 2(111)도 "111" 버킷에 들어가 그 버킷은 [11, 2]가 된다.
5(0111) 삽입: 상위 3비트 "011"은 아직 분할되지 않은 [8, 7] 버킷(지역 깊이 1, "0" 접두)으로 연결되어 있고 이미 가득 차 있어 분할한다. 지역 깊이(1)가 전역 깊이(3)보다 작으므로 디렉터리는 그대로 두고 버킷만 분할한다. 상위 2비트로 8(00)은 "00" 버킷에, 7(01)은 "01" 버킷에 들어간다. 5(01)도 "01" 버킷에 들어가 그 버킷은 [7, 5]가 된다.
13(0101) 삽입: 상위 3비트 "010"은 [7, 5] 버킷(지역 깊이 2, "01" 접두)으로 연결되어 있고 이미 가득 차 있어 분할한다. 지역 깊이(2)가 전역 깊이(3)보다 작으므로 디렉터리는 그대로 두고 버킷만 분할한다. 상위 3비트로 나누어 보면 7(011)과 5(011)이 공교롭게도 둘 다 "011"이 되어 같은 쪽에 남고, "010" 쪽 버킷은 비게 된다. 13(010)이 바로 이 빈 "010" 버킷에 들어가 그 버킷은 [13] 하나만 남는다.
최종적으로 인덱스 010은 [13](1개), 인덱스 011은 [7, 5](2개), 인덱스 111은 [11, 2](먼저 들어간 11이 첫 번째)가 된다.

NoteBase

### 선지 판단

① 키 값 7이 삽입될 때 새 버킷이 생성된다 → X. 7이 들어갈 때는 [8] 버킷에 여유가 있어 그대로 [8, 7]이 되었을 뿐 분할이나 새 버킷 생성이 일어나지 않았다.
② 모든 키 값이 삽입된 후 버킷의 개수는 5이다 → X. 최종 버킷은 [8], [4], [13], [7, 5], [9], [11, 2]로 총 6개이다.
③ 인덱스 010이 가리키는 버킷에는 2개의 키 값이 존재한다 → X. 인덱스 010이 가리키는 버킷에는 13 하나만 있다. 7과 5가 상위 3비트까지 같은 값(011)을 가져 둘 다 011 쪽에 남고 010 쪽은 비었다가, 이후 13이 그 자리에 홀로 들어갔기 때문이다.
④ 인덱스 111이 가리키는 버킷의 첫 번째 키 값은 11이다 → O. 인덱스 111이 가리키는 버킷은 [11, 2]이며, 9와 11이 있던 버킷이 분할될 때 11이 111 버킷에 먼저 자리 잡았고 이후 2가 그 뒤에 삽입되었으므로 첫 번째 키 값은 11이다.

### 오답 포인트

가장 헷갈리는 지점은 5번 삽입(13) 단계에서 [7, 5] 버킷을 상위 3비트로 나눌 때이다. 7(0110)과 5(0111)는 상위 3비트가 똑같이 011이어서 분할해도 한쪽 버킷(011)에 둘 다 남고 다른 쪽(010)은 빈 채로 남는다는 점을 놓치면, 이후 13이 010 버킷에 혼자 들어가는 것을 못 보고 010 버킷에 2개가 있다고 착각하기 쉽다(③의 함정). 또한 버킷 분할 시 디렉터리를 두 배로 늘려야 하는 경우(지역 깊이=전역 깊이)와 늘리지 않아도 되는 경우(지역 깊이<전역 깊이)를 혼동하면 최종 버킷 개수나 각 버킷의 내용이 달라진다.

### 다시 나오면 이렇게 판단

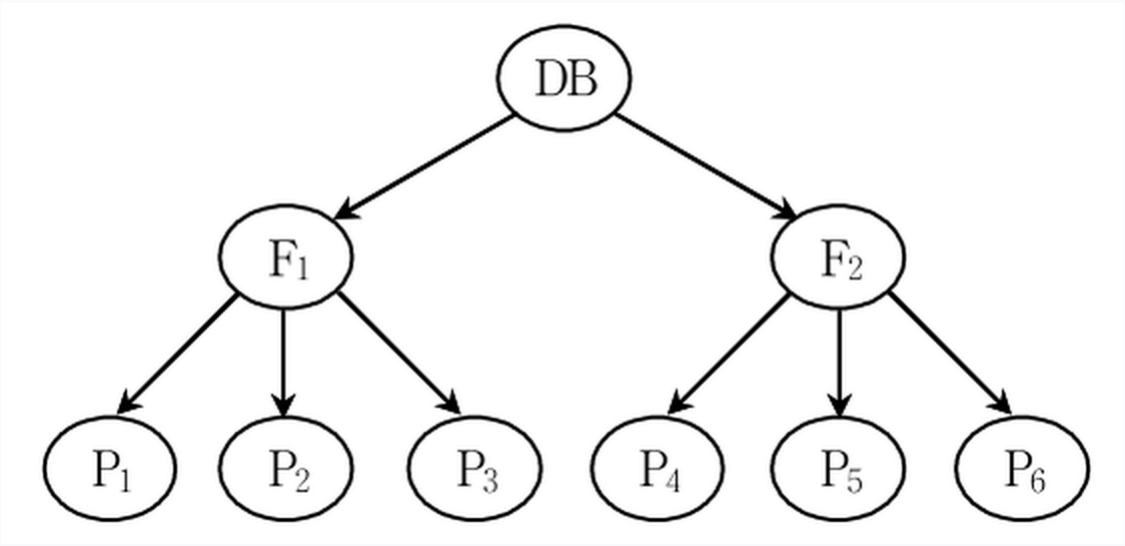
버킷이 가득 찬 상태에서 삽입이 일어날 때마다 반드시 그 버킷의 지역 깊이와 현재 디렉터리의 전역 깊이를 먼저 비교한다. 같으면 디렉터리를 두 배로 늘리고, 작으면 디렉터리는 그대로 두고 버킷만 분할한다고 판단한다. 분할 후에는 늘어난 비트 자리 값이 같은 키끼리 한쪽 버킷에 몰릴 수 있다는 점을 항상 의심하고, 실제 해시값 비트를 하나씩 직접 확인해 재배치한다.

### 암기 포인트

지역 깊이=전역 깊이이면 디렉터리 더블링, 지역 깊이<전역 깊이이면 버킷만 분할(디렉터리 불변). 삽입 순서대로 왼쪽부터 채운다는 조건 때문에 분할 후 버킷 내 키의 순서(첫 번째/두 번째)는 삽입된 시점 순서로 정해진다.

NoteBase

**문 17. 다음 그림은 다중 단위크기 로킹(multiple granularity locking) 기법에서 3계층 단위크기의 예를 보여주고 있다. 다중 단위크기 2단계 로킹 기법의 동작에 대한 설명으로 옳지 않은 것은? (단, 그림에서 DB는 F1과 F2로 구성되어 있고 F1은 P1, P2, P3으로 구성되어 있음을 의미하며, 로크 S는 공유(Shared), X는 배타(eXclusive), I는 의도(Intention)를 의미한다)**




① P3에 읽기 연산을 수행하기 위해서는 DB와 F1에 순서대로 IS 로크를 건 후 P3에 S 로크를 걸어야 한다.

② 트랜잭션 T1이 F1에 올바르게 S 로크를 걸고 읽기 작업을 진행하던 중 트랜잭션 T2가 P3에 쓰기 작업을 진행하려고 하면 로크를 걸지 못하여 대기해야 한다.

③ 트랜잭션 T1이 F1에 읽기 작업을 위해 로크를 거는 것과 트랜잭션 T2가 F2에 쓰기 작업을 위해 로크를 거는 것은 충돌이 발생하지 않는다.

④ 트랜잭션 T1이 P1, P4에 순서대로 작업을 해야할 때, P4와 관련된 로크를 걸기 전이라도 P1에 대한 작업을 완료하자마자 로크를 해제해주어 다른 트랜잭션들이 로크를 걸기 위해 대기하는 시간을 줄일 수 있다.

**정답 4번**

**출제 포인트**

다중 단위크기 로킹(multiple granularity locking)에서 의도 로크(IS/IX)의 계층적 부여 규칙과, 2단계 로킹(2PL) 프로토콜의 확장 국면(growing phase)·축소 국면(shrinking phase) 규칙을 함께 이해하는지를 묻는다. 그림의 트리는 DB 아래에 F1, F2가 있고 F1 아래에 P1, P2, P3, F2 아래에 P4, P5, P6이 있는 3계층 구조이며, 세부 항목(P)에 로크를 걸려면 그 조상 노드들에 먼저 의도 로크를 걸어야 한다는 점과, 2PL은 한 트랜잭션이 로크를 하나라도 해제하고 나면 그 이후에는 새 로크를 추가로 걸 수 없다는 점이 핵심이다.

NoteBase

### 올바른 순서

다중 단위크기 로킹에서 세부 데이터(P)에 접근하려면 반드시 루트(DB)에서부터 시작해 최상위 조상 → 하위 조상 → 대상 노드 순으로 로크를 건다. 읽기 연산이면 조상 노드에 의도 공유(IS) 로크를 걸고 최종적으로 대상 노드에 S 로크를 건다. 쓰기 연산이면 조상 노드에 의도 배타(IX) 로크를 걸고 최종적으로 대상 노드에 X 로크를 건다. 2단계 로킹은 한 트랜잭션 안에서 로크 개수가 늘어나기만 하는 확장 국면과 이후 줄어들기만 하는 축소 국면으로 나뉘며, 축소 국면에 진입한(즉 로크를 하나라도 해제한) 트랜잭션은 그 뒤로 새로운 로크를 추가로 취득할 수 없다.

### 선지별 해설

① O: P3은 F1의 자식이고 F1은 DB의 자식이므로, P3에 읽기(S) 로크를 걸려면 그 조상인 DB와 F1에 순서대로 의도 공유(IS) 로크를 먼저 걸어야 한다. 이는 다중 단위크기 로킹의 기본 규칙과 일치하는 옳은 설명이다.
② O: T1이 F1에 S 로크를 보유한 채 읽기 작업 중일 때, T2가 F1의 자식인 P3에 쓰기 작업을 하려면 먼저 F1에 IX 로크를 걸어야 한다. 그런데 S(공유)와 IX(의도 배타)는 서로 호환되지 않는 로크 쌍이므로 T2는 F1에 IX 로크를 걸 수 없어 대기해야 한다. 옳은 설명이다.
③ O: F1과 F2는 DB 아래의 서로 다른 하위 트리(형제 노드)로, 조상-자손 관계가 아니다. T1이 F1 계열에 읽기 로크를 걸고 T2가 F2 계열에 쓰기 로크를 거는 것은 서로 겹치는 노드가 없으므로 충돌이 발생하지 않는다. 옳은 설명이다.
④ X: P1에 대한 작업을 마치자마자 그 로크를 바로 해제하면, 그 순간부터 T1은 축소 국면(shrinking phase)에 들어간 것이 된다. 2단계 로킹 규칙상 축소 국면에 진입한 트랜잭션은 이후 어떤 새로운 로크도 추가로 취득할 수 없으므로, 뒤이어 P4 작업을 위해 필요한 로크(및 그 조상인 F2에 대한 의도 로크)를 걸 수 없게 되어 위반이 발생한다. 따라서 이 설명은 2단계 로킹 기법의 규칙에 어긋나는 옳지 않은 설명이다.

### 오답 포인트

①②③은 모두 다중 단위크기 로킹의 의도 로크 부여 규칙과 로크 호환성 규칙을 정확히 따른 설명이어서 옳다고 판단해야 한다. 자칫 ②를 'S와 IX는 관련이 없어 보이니 충돌이 아니다'라고 오해하기 쉬운데, S와 IX는 호환되지 않는 로크 쌍이므로 대기가 맞다. ④는 로크를 '일찍 풀어주면 효율적'이라는 상식적인 생각과 반대로, 2PL에서는 로크를 하나라도 해제하는 순간 그 트랜잭션은 축소 국면에 들어가 이후 어떤 로크도 새로 취득할 수 없다는 점이 함정이다.

### 다시 나오면 이렇게 판단

다중 단위크기 로킹 문제는 먼저 트리 구조에서 대상 노드의 조상이 무엇인지 확인하고, 읽기면 IS, 쓰기면 IX를 조상에 순서대로 건 뒤 대상 노드에 S 또는 X를 거는 규칙을 적용한다. 로크 충돌 여부는 형제 관계(공통 조상만 있고 서로 겹치지 않음)인지, 아니면 한쪽이 다른 쪽의 조상-자손인지를 먼저 구분해서 판단한다. 2PL 관련 선지는 '로크를 언제 거는가'가 아니라 '로크를 언제 푸는가'와 '푼 뒤에 새 로크를 걸려고 하는가'를 확인해야 한다 — 해제 이후 신규 취득이 있으면 무조건 규칙 위반이다.

NoteBase

**암기 포인트**

다중 단위크기 로킹: 세부 항목에 로크를 걸려면 조상에 먼저 의도 로크(IS/IX)를 건다. 로크 호환성: S–IX는 호환되지 않는다(공유 중인 노드에 하위 배타 작업을 시도하면 대기). 형제 하위 트리 간 로크는 서로 충돌하지 않는다. 2단계 로킹: 확장 국면(취득만)과 축소 국면(해제만)이 있고, 한 번이라도 해제하면 그 즉시 축소 국면 진입 — 이후 신규 로크 취득은 모두 규칙 위반이다.

NoteBase

**문 18. 데이터베이스 관리자 DBA가 생성한 테이블 T에 대해 사용자 USR1이 T에 대한 뷰 V를 생성한 후 사용자 USR2가 V에 대한 검색을 실행할 수 있도록 할 때, 반드시 필요한 권한부여 작업만을 모두 나열한 것은?**

ㄱ. DBA가 "GRANT CREATE VIEW TO USR1"을 실행

ㄴ. DBA가 "GRANT SELECT ON T TO USR1"을 실행

ㄷ. DBA가 "GRANT SELECT ON T TO USR1 WITH GRANT OPTION"을 실행

ㄹ. USR1이 "GRANT SELECT ON T TO USR2"를 실행

ㅁ. USR1이 "GRANT SELECT ON V TO USR2"를 실행

① ㄴ, ㅁ

② ㄱ, ㄷ, ㄹ

③ ㄱ, ㄷ, ㅁ

④ ㄴ, ㄷ, ㄹ

**정답 3번**

**출제 포인트**

테이블에 대한 뷰를 새로 만들고, 그 뷰를 다른 사용자에게 다시 권한부여(GRANT)할 수 있으려면 어떤 권한들이 사슬처럼 연결되어야 하는지를 묻는 문제이다. 뷰 생성 권한, 기본 테이블에 대한 접근 권한, 그리고 그 권한을 남에게 넘겨줄 수 있는 GRANT OPTION이 각각 다른 역할을 한다는 점이 핵심이다.

**핵심 개념**

DBA가 만든 테이블 T 위에 USR1이 뷰 V를 만들려면, 첫째 USR1에게 뷰를 만들 수 있는 시스템 권한(CREATE VIEW)이 있어야 하고, 둘째 뷰의 정의에 사용되는 테이블 T에 대한 SELECT 권한이 있어야 한다. 그런데 USR1이 만든 뷰 V에 대한 SELECT 권한을 USR2에게 다시 부여하려면, USR1이 뷰의 근거가 되는 테이블 T에 대해 단순 SELECT 권한만 가진 것으로는 부족하고 GRANT OPTION(권한을 다른 사람에게 넘겨줄 수 있는 권한)까지 가지고 있어야 한다. GRANT OPTION 없이 부여받은 권한은 그 권한으로 만든 뷰에 대해서도 타인에게 재부여할 수 없기 때문이다.

NoteBase

### 선지별 해설

ㄱ. DBA가 "GRANT CREATE VIEW TO USR1"을 실행 → O. USR1이 뷰 V를 생성하려면 뷰를 만들 수 있는 시스템 권한이 먼저 있어야 하므로 반드시 필요하다.
ㄴ. DBA가 "GRANT SELECT ON T TO USR1"을 실행 → X. USR1이 테이블 T를 조회해 뷰를 만드는 데는 쓰이지만, GRANT OPTION이 없어 USR1이 이 권한에 기반한 뷰 V의 SELECT 권한을 USR2에게 다시 부여할 수 없다. 따라서 이 형태로는 목적을 달성할 수 없고, ㄷ으로 대체되어야 한다.
ㄷ. DBA가 "GRANT SELECT ON T TO USR1 WITH GRANT OPTION"을 실행 → O. USR1이 테이블 T에 대한 SELECT 권한을 GRANT OPTION과 함께 받아야, 이를 바탕으로 만든 뷰 V에 대한 SELECT 권한을 USR2에게 넘겨줄 수 있다. 반드시 필요하다.
ㄹ. USR1이 "GRANT SELECT ON T TO USR2"를 실행 → X. USR2는 뷰 V만 조회하면 되고 테이블 T에 직접 접근할 필요가 없으므로 불필요한 작업이다.
ㅁ. USR1이 "GRANT SELECT ON V TO USR2"를 실행 → O. USR2가 실제로 뷰 V를 검색할 수 있으려면 USR1이 뷰 V에 대한 SELECT 권한을 USR2에게 직접 부여해야 한다. 반드시 필요하다.

### 오답 포인트

ㄴ과 ㄷ을 혼동하기 쉬운데, 단순 SELECT 권한(ㄴ)과 GRANT OPTION이 붙은 SELECT 권한(ㄷ)의 차이가 이 문제의 핵심이다. USR1이 단지 테이블 T를 읽을 수 있는 것과, 그 권한을 근거로 만든 뷰의 권한을 남에게 다시 넘겨줄 수 있는 것은 별개의 자격이다. 또한 ㄹ처럼 USR2에게 원본 테이블 T의 권한까지 주는 것은 문제에서 요구하는 목적(뷰 V 검색)을 넘어서는 불필요한 작업이라는 점도 놓치기 쉽다.

### 다시 나오면 이렇게 판단

뷰를 만드는 사용자가 다시 그 뷰의 권한을 제3자에게 넘겨주는 시나리오가 나오면, 뷰 생성 권한(CREATE VIEW)과 기본 테이블에 대한 권한을 나누어 확인하고, 기본 테이블 권한에 GRANT OPTION이 붙어 있는지부터 따진다. GRANT OPTION이 없으면 그 사용자는 자신이 만든 뷰에 대한 권한을 남에게 다시 부여할 수 없다는 원칙을 기준으로 필요한 GRANT 문만 골라낸다.

### 암기 포인트

뷰 생성 = CREATE VIEW 권한 + 기본 테이블 SELECT 권한. 뷰 권한의 재부여 = 기본 테이블 권한에 WITH GRANT OPTION 필수. 최종 사용자에게는 뷰에 대한 권한만 부여하면 충분하고 기본 테이블 권한까지 줄 필요는 없다.

NoteBase

**문 19. 다음 그림은 ARIES 회복 기법에 의해 시스템 고장 전에 마지막으로 기록된 DBMS 로그이다. 이 로그를 사용하여 수행하는 회복 과정에 대한 설명으로 옳지 않은 것은? (단, LSN은 로그 순차번호(Log Sequence Number)이며, 표현되지 않은 로그의 다른 속성은 고려하지 않는다)**

| LSN | 최근 이전 LSN | 트랜잭션 번호 | 연산 종류 | 페이지 번호 |
|---|---|---|---|---|
| 1 | 0 | $T_1$ | 갱신 | $P_1$ |
| 2 | 0 | $T_2$ | 갱신 | $P_2$ |
| 3 | 1 | $T_1$ | 완료 | |
| 4 | begin checkpoint | | | |
| 5 | end checkpoint | | | |
| 6 | 0 | $T_3$ | 갱신 | $P_1$ |

① 분석 단계에서 트랜잭션 T2와 T3은 처리 중(in progress)인 트랜잭션으로 분류된다.

② REDO 단계에서 오손 페이지 테이블(dirty page table)을 사용하여 불필요한 REDO 연산의 수행을 줄여 회복시간을 단축할 수 있다.

③ UNDO 단계에서 완료되지 않은 트랜잭션의 복귀(rollback) 시 수행한 갱신연산에 해당하는 UNDO에 대해 보상 로그 레코드(compensation log record)를 기록하지 않는다.

④ UNDO 단계에서는 로그의 끝에서부터 반대 방향으로 완료되지 않은 트랜잭션에서 수행된 갱신 연산을 UNDO한다.

**정답 3번**

**출제 포인트**

ARIES 회복 기법의 세 단계(분석-REDO-UNDO)가 로그를 어떻게 활용하는지 묻는 문항이다. 주어진 로그 표에서 각 트랜잭션의 상태와 체크포인트 위치를 읽어내고, 그 정보로 분석·REDO·UNDO 단계가 실제로 어떤 동작을 하는지를 판단해야 한다.

NoteBase

### 자료 읽는 순서

먼저 로그를 LSN 순서대로 훑어 각 트랜잭션의 마지막 상태를 확인한다. T1은 LSN1에서 P1을 갱신하고 LSN3에서 완료(commit) 로그를 남겼으므로 고장 시점에 이미 종료된 트랜잭션이다. T2는 LSN2에서 P2를 갱신한 뒤 완료 로그가 없으므로 고장 시점까지 진행 중인 트랜잭션이다. T3은 LSN6에서 P1을 갱신한 뒤 완료 로그가 없으므로 역시 진행 중인 트랜잭션이다. LSN4, LSN5는 체크포인트 시작과 종료를 나타내는 로그로, 분석 단계가 시작되는 기준점이 된다. 이 순서로 정리하면 분석 단계에서 트랜잭션 테이블에는 T2, T3만 진행 중 상태로 남고, UNDO 단계에서 되돌려야 할 대상도 T2와 T3의 갱신 연산이 된다.

### 선지별 해설

① O - 분석 단계는 로그를 훑어 트랜잭션 테이블을 재구성하는데, T1은 LSN3에서 완료 로그가 있으므로 제외되고, T2와 T3은 완료 로그가 없으므로 고장 시점에 처리 중(in progress)인 트랜잭션으로 분류된다.
② O - REDO 단계는 오손 페이지 테이블(dirty page table)에 기록된 각 페이지의 recLSN(해당 페이지에 반영되지 않은 갱신이 처음 발생한 LSN)을 이용해 REDO를 시작할 최소 LSN을 정한다. 이 정보를 활용하면 이미 디스크에 반영된 갱신까지 처음부터 다시 수행할 필요가 없어 불필요한 REDO를 줄이고 회복 시간을 단축할 수 있다.
③ X - UNDO 단계는 완료되지 않은 트랜잭션(T2, T3)의 갱신 연산을 되돌리면서, 되돌린 연산마다 보상 로그 레코드(CLR)를 반드시 기록한다. CLR은 되돌린 내용을 다시 로그로 남겨, 회복 도중 두 번째 고장이 발생해도 이미 UNDO한 연산을 중복해서 되돌리지 않도록 해 준다. 따라서 “보상 로그 레코드를 기록하지 않는다”는 설명은 틀린 표현이며, 올바른 표현은 “보상 로그 레코드를 기록한다”이다.
④ O - UNDO 단계는 로그의 끝(가장 최근 LSN)에서부터 이전 LSN 방향으로 거슬러 올라가며, 완료되지 않은 트랜잭션(T2, T3)이 수행한 갱신 연산만 골라 역순으로 UNDO한다.

### 오답 포인트

①, ②, ④를 정답으로 고르는 실수는 REDO와 UNDO 단계의 역할을 서로 바꿔 생각하거나, 오손 페이지 테이블과 보상 로그 레코드의 목적을 혼동할 때 발생한다. 오손 페이지 테이블은 REDO 범위를 줄이는 용도이고, 보상 로그 레코드는 UNDO 과정 자체를 로그로 남겨 재고장에 대비하는 용도라는 점을 구분해야 한다. ③은 이 CLR 기록 여부를 반대로 서술한 함정이다.

### 다시 나오면 이렇게 판단

ARIES 문제에서 로그 표가 주어지면, 완료 로그가 있는 트랜잭션은 분석 단계에서 제외되고 없는 트랜잭션만 UNDO 대상이 된다는 점을 먼저 확정한다. 이어서 선지에 “보상 로그 레코드를 기록하지 않는다”처럼 UNDO 과정에서 CLR을 생략한다는 서술이 나오면, ARIES의 UNDO는 항상 CLR을 남긴다는 원칙에 따라 이를 옳지 않은 설명으로 판단한다.

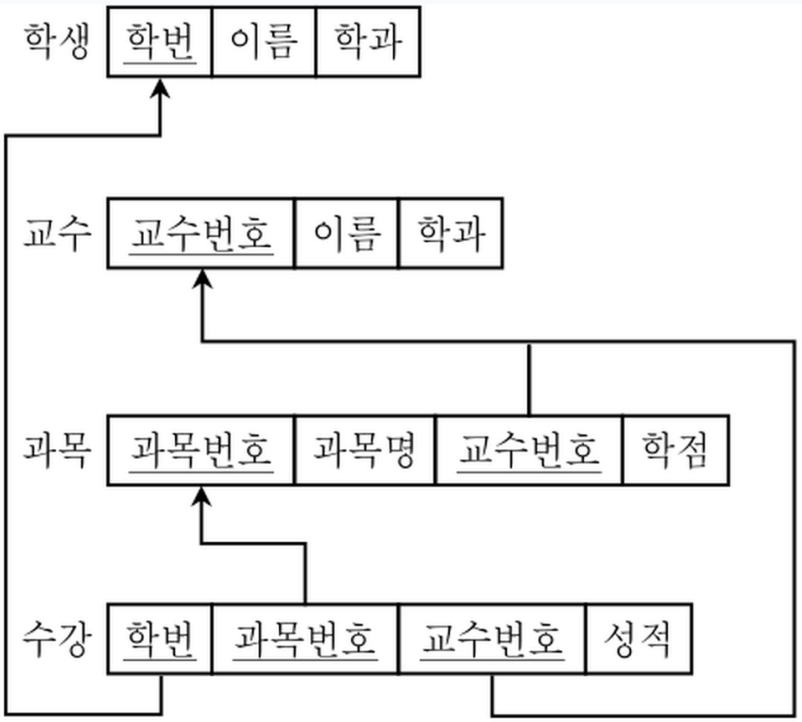
NoteBase

**암기 포인트**

ARIES 3단계: 분석(트랜잭션 테이블·오손 페이지 테이블 재구성) → REDO(오손 페이지 테이블의 recLSN을 이용해 필요한 갱신만 재수행) → UNDO(로그 끝에서 역순으로 미완료 트랜잭션의 갱신을 되돌리며 매 UNDO마다 보상 로그 레코드를 기록).

NoteBase

**문 20. 다음 그림은 학생, 교수, 과목, 수강 4개의 테이블 스키마와 그 인스턴스를 보여주고 있다. 이 데이터베이스에 SQL문을 수행하였을 때, 오류가 발생하지 않는 것은? (단, 밑줄은 기본키를 의미한다)**




학생

| 학번 | 이름 | 학과 |
|---|---|---|
| 100 | 강감찬 | 컴퓨터 |
| 200 | 안중근 | 컴퓨터 |
| 300 | 이순신 | 수학 |
| 400 | 김구 | 전자 |

교수

| 교수번호 | 이름 | 학과 |
|---|---|---|
| 1000 | 손기훈 | 컴퓨터 |
| 2000 | 남승민 | 컴퓨터 |
| 3000 | 양정상 | 수학 |
| 4000 | 장은상 | 전자 |
| 5000 | 김연호 | 경영 |

NoteBase

과목

| 과목번호 | 과목명 | 교수번호 | 학점 |
|---|---|---|---|
| C001 | 데이터베이스 | 1000 | 3 |
| C001 | 데이터베이스 | 2000 | 3 |
| C002 | 자료구조 | 1000 | 3 |
| M001 | 이산수학 | 3000 | 3 |
| E001 | 회로이론 | 4000 | 3 |

수강

| 학번 | 과목번호 | 교수번호 | 성적 |
|---|---|---|---|
| 100 | C001 | 1000 | A+ |
| 200 | C001 | 2000 | B+ |
| 200 | C002 | 1000 | A |
| 300 | M001 | 3000 | C+ |
| 400 | E001 | 4000 | B |

① 
```
UPDATE 교수 SET 교수번호 = 2000
  WHERE 이름 = '김연호';
```

② 
```
CREATE VIEW 컴퓨터학과
  AS SELECT 이름, 학과
  FROM 학생
  WHERE 학과 = '컴퓨터';
INSERT INTO 컴퓨터학과
  VALUES ('윤봉길', '컴퓨터');
```

③ 
```
CREATE VIEW 교수별수강학생(교수번호, 학번, 과목번호, 성적)
  AS SELECT 교수.교수번호, 학번, 과목번호, 성적
  FROM 교수, 수강
  WHERE 교수.교수번호 = 수강.교수번호;
INSERT INTO 교수별수강학생(학번, 과목번호, 성적)
  VALUES ('300','C002', 'B');
```

④ 
```
CREATE VIEW 교수명단
  AS SELECT 교수번호, 이름
  FROM 교수;
INSERT INTO 교수명단
  VALUES (6000, '진상오');
```

정답 4번

NoteBase

### 출제 포인트

기본키 제약 조건(중복·NULL 금지)과 뷰(view)에 대한 삽입(INSERT)이 언제 성공하고 언제 실패하는지를 함께 묻는 문항이다. 특히 뷰가 단일 테이블에서 파생되었는지 여러 테이블의 조인으로 파생되었는지, 그리고 뷰에 포함되지 않은 속성이 기본키인지 아닌지에 따라 삽입 성공 여부가 갈린다.

### 핵심 개념

① 기본키 무결성: 기본키 값은 중복될 수 없고 NULL이 될 수 없다. ② 뷰를 통한 삽입: 하나의 기본 테이블에서 SELECT로 정의된 뷰(조인·집계 없음)는 원칙적으로 삽입이 가능하지만, 뷰에 포함되지 않은 속성 중 기본키처럼 값이 반드시 있어야 하는 속성이 있으면 그 속성 값을 채울 수 없어 삽입이 실패한다. ③ 여러 테이블을 조인하여 정의한 뷰는 갱신(삽입·수정·삭제)이 불가능한 것이 일반적이다. 조인 결과의 각 행이 어느 기본 테이블의 어느 행에서 온 것인지 하나로 대응시킬 수 없기 때문이다.

### 선지별 해설

① UPDATE 교수 SET 교수번호 = 2000 WHERE 이름 = '김연호'; → 오류 발생. 김연호의 교수번호(5000)를 2000으로 바꾸려 하는데, 교수번호는 교수 테이블의 기본키이고 2000은 이미 남승민의 교수번호로 사용 중이다. 기본키 값이 중복되므로 기본키 무결성 제약을 위반하여 오류가 발생한다.
② CREATE VIEW 컴퓨터학과 AS SELECT 이름, 학과 FROM 학생 WHERE 학과 = '컴퓨터'; INSERT INTO 컴퓨터학과 VALUES ('윤봉길', '컴퓨터'); → 오류 발생. 이 뷰는 학생 테이블 하나에서 파생되어 겉보기에는 삽입이 가능해 보이지만, 뷰에는 학생 테이블의 기본키인 학번이 포함되어 있지 않다. 삽입 시 학번 값을 채울 방법이 없는데 학번은 기본키라 NULL을 허용하지 않으므로 삽입이 실패한다.
③ CREATE VIEW 교수별수강학생(교수번호, 학번, 과목번호, 성적) AS SELECT 교수.교수번호, 학번, 과목번호, 성적 FROM 교수, 수강 WHERE 교수.교수번호 = 수강.교수번호; INSERT INTO 교수별수강학생(학번, 과목번호, 성적) VALUES ('300', 'C002', 'B'); → 오류 발생. 이 뷰는 교수 테이블과 수강 테이블 두 개를 조인하여 정의되었다. 조인으로 만들어진 뷰는 삽입되는 한 행이 교수 테이블과 수강 테이블 중 어느 기본 테이블의 어느 행에 반영되어야 하는지 대응시킬 수 없으므로 삽입 자체가 허용되지 않아 오류가 발생한다.
④ CREATE VIEW 교수명단 AS SELECT 교수번호, 이름 FROM 교수; INSERT INTO 교수명단 VALUES (6000, '진상오'); → 오류 없음. 이 뷰는 교수 테이블 하나에서 파생되었고, 삽입되는 값 (6000, '진상오')은 뷰에 포함된 교수번호와 이름에 각각 대응된다. 교수번호 6000은 기존 값과 중복되지 않아 기본키 제약에 위배되지 않고, 뷰에 포함되지 않은 학과 속성은 기본키가 아니므로 NULL로 남아도 무방하다. 따라서 삽입이 정상적으로 수행된다.

NoteBase

### 오답 포인트

②와 ④를 혼동하기 쉽다. 둘 다 단일 테이블에서 파생된 뷰에 삽입하는 상황이지만, ②는 뷰에서 빠진 속성이 기본키(학번)이고 ④는 뷰에서 빠진 속성이 기본키가 아닌 일반 속성(학과)이라는 차이가 있다. 뷰에 없는 속성이 기본키이면 반드시 값이 있어야 하므로 삽입이 실패하고, 기본키가 아니면 NULL로 두어도 무방하므로 삽입이 성공한다는 점을 구분해야 한다. 또한 ③처럼 여러 테이블의 조인으로 정의된 뷰는 애초에 갱신 대상이 될 수 없다는 점도 놓치기 쉽다.

### 다시 나오면 이렇게 판단

SQL 문이 오류를 일으키는지 묻는 문항에서는 먼저 기본키 중복·NULL 여부를 확인하고, 뷰가 등장하면 그 뷰가 몇 개의 테이블에서 파생되었는지부터 확인한다. 조인으로 만든 뷰라면 삽입·수정·삭제가 불가능하다고 바로 판단하고, 단일 테이블에서 만든 뷰라면 뷰에 포함되지 않은 속성 중 기본키가 있는지를 확인해 그 값이 채워질 수 있는지를 따진다.

### 암기 포인트

여러 테이블을 조인한 뷰는 갱신 불가. 단일 테이블 뷰라도 뷰에서 빠진 속성이 기본키이면 삽입 불가, 빠진 속성이 일반 속성이면 삽입 가능. 기본키는 중복과 NULL을 모두 허용하지 않는다.

NoteBase