BACKEND CS · STUDY GUIDE

DB·쿼리 최적화 9강

정규화 · 인덱스 · 실행 계획 · 캐시 · 페이지네이션

9강 · 3부 구성

ttukttak-coding.dev

2026

이 책은

이 책은 데이터베이스 설계와 쿼리 성능을 한 권에 모았습니다. 정규화로 시작해 인덱스, 실행 계획, 커넥션 풀, N+1, 캐시, 페이지네이션까지 — 조회를 느리게 만드는 원인과 구조적인 해결책을 다룹니다.

각 강의는 "왜 느려지는가?" → "어떻게 구조를 바꾸는가?" → "무엇을 측정해 확인하는가?" 의 순서로 풀어갑니다. 쿼리 한 줄 튜닝이 아니라, 같은 패턴이 반복될 때 적용 가능한 사고 틀을 만드는 데 목표가 있습니다.

1부에서는 설계의 출발점인 정규화, 2부에서는 인덱스와 실행 계획, 3부에서는 운영 단계의 조회 성능 이슈를 다룹니다. 한 번 통독한 뒤에는 새 쿼리를 짤 때 옆에 두고 점검 도구로 쓰셔도 좋습니다.

> **구성**: 3부 9강 / **대상**: 쿼리 성능을 구조로 풀고 싶은 백엔드 개발자 / **블로그**: 더 많은 글은 **ttukttak-coding.dev** 에서 보실 수 있습니다.

목차

1부. 데이터베이스 설계 기초

2부. 인덱스와 실행 계획

3부. 조회 성능 — 커넥션 풀 · 캐시 · 페이지네이션

PART 1

1부. 데이터베이스 설계 기초

Chapter 1

데이터베이스 정규화 완전 정복 — 1NF부터 BCNF까지

#정규화

정규화가 왜 필요한지, 각 단계가 어떤 이상 현상을 해결하는지 예제 테이블과 함께 정리합니다.

정규화, 왜 해야 하나요?

테이블 설계를 대충 하면 어떤 일이 벌어질까요?

- **삽입 이상** — 강의를 아직 수강하지 않은 학생 정보를 넣을 수 없습니다
- **갱신 이상** — 교수 이름이 바뀌면 수백 행을 전부 수정해야 합니다
- **삭제 이상** — 수강 기록을 지웠더니 학생 정보까지 사라집니다

이 세 가지를 **이상 현상(Anomaly)** 이라 부르며, 정규화는 이 이상 현상을 단계적으로 제거하는 과정입니다. 각 단계(Normal Form)가 어떤 문제를 해결하는지, 비정규 테이블 하나를 끝까지 분해하면서 살펴보겠습니다.

먼저 알아야 할 용어들

정규화를 이해하려면 몇 가지 용어를 먼저 짚고 넘어가야 합니다.

함수적 종속 (Functional Dependency)

`X`의 값이 정해지면 `Y`의 값이 **하나로 결정**되는 관계를 `X → Y`라고 쓰고, "Y는 X에 함수적으로 종속된다"라고 합니다.

```
학번 → 이름
```

`학번`이 `S001`이면 `이름`은 항상 `김철수`입니다. 학번을 알면 이름이 하나로 결정되므로 `이름`은 `학번`에 함수적으로 종속됩니다. 여기서 `학번`처럼 값을 결정하는 쪽을 **결정자(Determinant)**, `이름`처럼 결정되는 쪽을 **종속자(Dependent)** 라고 합니다.

완전 함수 종속 (Full Functional Dependency)

기본키의 **모든 속성을 사용해야만** 종속자가 결정되는 관계입니다.

```
(학번, 과목코드) → 성적
```

`성적`은 `학번`만으로도, `과목코드`만으로도 알 수 없습니다. 두 속성이 모두 있어야 결정되므로 **완전 함수 종속**입니다.

부분 함수 종속 (Partial Functional Dependency)

복합 기본키의 **일부 속성만으로** 종속자가 결정되는 관계입니다.

```
(학번, 과목코드) → 이름      -- 실제로는 학번만으로 이름이 결정됨
```

기본키가 `(학번, 과목코드)`인데, `이름`은 `학번`만 알면 결정됩니다. 기본키의 일부에만 종속되어 있으므로 **부분 함수 종속**이고, 이것이 **2NF에서 제거해야 할 대상**입니다.

이행적 함수 종속 (Transitive Functional Dependency)

`X → Y → Z`처럼, 기본키가 아닌 속성을 **거쳐서** 다른 속성이 결정되는 관계입니다.

```
학번 → 지도교수 → 연구실
```

`연구실`은 `학번`에 직접 종속된 것이 아니라 `지도교수`를 경유해서 결정됩니다. 이것이 **이행적 함수 종속**이고, **3NF에서 제거해야 할 대상**입니다.

후보키와 결정자

- **후보키(Candidate Key)** — 테이블에서 각 행을 유일하게 식별할 수 있는 **최소한의 속성 집합**입니다. 후보키가 여러 개일 수 있고, 그중 하나를 기본키로 선택합니다
- **결정자(Determinant)** — 함수 종속에서 다른 속성의 값을 결정하는 속성입니다. `X → Y`에서 `X`가 결정자입니다

BCNF에서는 **모든 결정자가 후보키**여야 합니다. 결정자인데 후보키가 아닌 속성이 있으면 BCNF를 위반합니다.

> **참고:** 이 용어들은 아래 각 정규형 설명에서 반복적으로 등장합니다. 읽다가 헷갈리면 여기로 돌아와 확인하세요.

출발점: 비정규 테이블

예제로 사용할 수강 관리 테이블입니다.

학번	이름	학과	과목코드	과목명	교수	성적
S001	김철수	컴퓨터공학	CS101	자료구조	박교수	A
S001	김철수	컴퓨터공학	CS102	운영체제	이교수	B+
S002	이영희	전자공학	CS101	자료구조	박교수	A+
S002	이영희	전자공학	EE201	회로이론	최교수	B

한눈에 봐도 `이름`, `학과`, `과목명`, `교수`가 중복되고 있습니다. 이 테이블을 단계별로 정리해 보겠습니다.

Phase 1. 제1정규형 (1NF)

규칙: 모든 컬럼 값은 원자값(Atomic Value)이어야 한다

1NF는 가장 기본적인 규칙입니다. 하나의 셀에 여러 값이 들어가면 안 됩니다.

1NF 위반 예시:

학번	이름	과목코드
S001	김철수	CS101, CS102

위처럼 하나의 셀에 여러 과목코드가 들어가면 `WHERE 과목코드 = 'CS101'` 같은 쿼리가 정상 동작하지 않습니다. 행을 분리해서 각 셀이 하나의 값만 갖도록 만들면 **1NF를 충족**합니다.

우리 예제 테이블은 이미 행이 분리되어 있으므로 1NF를 만족합니다.

> **참고:** 1NF의 핵심은 **"반복 그룹 제거"**입니다. 배열이나 콤마 구분 문자열이 컬럼에 들어가 있다면 1NF 위반을 의심하세요.

Phase 2. 제2정규형 (2NF)

규칙: 부분 함수 종속을 제거한다

2NF는 **복합 기본키**를 사용하는 테이블에서 의미가 있습니다. 기본키가 `(학번, 과목코드)`라고 했을 때, 기본키의 **일부분에만** 종속되는 컬럼이 있으면 안 됩니다.

문제: 부분 함수 종속이 뭔가요?

현재 테이블의 함수 종속(FD)을 나열하면 이렇습니다.

```
(학번, 과목코드) → 성적          -- 기본키 전체에 종속 ✅
학번 → 이름, 학과                 -- 기본키의 일부(학번)에만 종속 ❌
과목코드 → 과목명, 교수           -- 기본키의 일부(과목코드)에만 종속 ❌
```

`이름`과 `학과`는 `과목코드`와 관계없이 `학번`만으로 결정됩니다. 이것이 **부분 함수 종속**이고, 갱신 이상의 원인입니다.

해결: 테이블 분리

부분 종속되는 컬럼들을 별도 테이블로 분리합니다.

학생 테이블

학번	이름	학과
S001	김철수	컴퓨터공학
S002	이영희	전자공학

과목 테이블

과목코드	과목명	교수
CS101	자료구조	박교수
CS102	운영체제	이교수
EE201	회로이론	최교수

수강 테이블

학번	과목코드	성적
S001	CS101	A
S001	CS102	B+
S002	CS101	A+
S002	EE201	B

이제 김철수의 학과가 바뀌어도 **학생 테이블 한 행만** 수정하면 됩니다. 2NF 달성입니다.

> **참고:** 기본키가 단일 컬럼이면 부분 종속 자체가 발생하지 않으므로 자동으로 2NF를 만족합니다.

Phase 3. 제3정규형 (3NF)

규칙: 이행적 함수 종속을 제거한다

2NF까지 분리한 **과목 테이블**을 다시 보겠습니다.

```
과목코드 → 과목명
과목코드 → 교수
```

여기까지는 문제가 없어 보이지만, 만약 "하나의 과목은 반드시 한 명의 교수가 담당한다"는 규칙 대신 "한 교수는 한 과목만 담당한다"는 규칙이 있다면 어떨까요?

```
과목코드 → 교수
교수 → 과목코드
```

이 경우는 괜찮습니다. 하지만 다음과 같은 경우를 생각해 보겠습니다.

문제: A → B → C 이행 종속

학생 테이블에 `지도교수`와 `지도교수_연구실`이 추가된다고 가정합니다.

학번	이름	학과	지도교수	지도교수_연구실
S001	김철수	컴퓨터공학	박교수	공학관 301
S002	이영희	전자공학	최교수	공학관 502
S003	박민수	컴퓨터공학	박교수	공학관 301

함수 종속을 보면:

```
학번 → 지도교수 → 지도교수_연구실
```

`지도교수_연구실`은 기본키(`학번`)에 직접 종속된 것이 아니라 `지도교수`를 거쳐 **이행적으로 종속**됩니다. 박교수의 연구실이 이전하면 S001, S003 두 행 모두 수정해야 합니다.

해결: 이행 종속 분리

학생 테이블:

학번	이름	학과	지도교수
S001	김철수	컴퓨터공학	박교수
S002	이영희	전자공학	최교수
S003	박민수	컴퓨터공학	박교수

교수 테이블:

교수	연구실
박교수	공학관 301
최교수	공학관 502

이제 연구실 정보는 **교수 테이블 한 곳**에서만 관리합니다. 3NF 달성입니다.

> **참고:** 3NF의 공식 정의는 "비주요 속성(non-prime attribute)이 어떤 **후보키**에도 **비이행적으로** 종속"입니다. 쉽게 말해, **후보키가 아닌 컬럼이 다른 비키 컬럼을 결정하면 안 된다**는 뜻입니다.

Phase 4. 보이스-코드 정규형 (BCNF)

규칙: 모든 결정자가 후보키여야 한다

3NF는 "기본키가 아닌 속성" 사이의 종속만 제거합니다. **BCNF**는 한 단계 더 나아가 **모든 함수 종속의 결정자가 후보키**여야 한다고 요구합니다.

문제: 후보키가 아닌 결정자

수강 지도 테이블을 예로 들겠습니다. "한 학생은 과목당 한 명의 지도교수에게 배우고, 각 교수는 하나의 과목만 지도한다"는 규칙이 있다고 가정합니다.

학번	과목명	교수
S001	자료구조	박교수
S001	운영체제	이교수
S002	자료구조	박교수

후보키: `(학번, 과목명)` 또는 `(학번, 교수)`

함수 종속:

```
(학번, 과목명) → 교수      ✅ 결정자가 후보키
교수 → 과목명              ❌ 결정자(교수)가 후보키가 아님!
```

`교수`가 `과목명`을 결정하는데, `교수`는 후보키가 아닙니다. 이 상태에서 "정교수가 자료구조를 담당한다"는 사실을 저장하려면 학생이 수강 신청할 때까지 기다려야 합니다 — **삽입 이상**입니다.

해결: 결정자를 기본키로 분리

교수-과목 테이블:

교수	과목명
박교수	자료구조
이교수	운영체제

수강지도 테이블:

학번	교수
S001	박교수
S001	이교수
S002	박교수

모든 결정자가 후보키가 되었습니다. BCNF 달성입니다.

한눈에 보는 정규형 비교

정규형	핵심 규칙	제거 대상
1NF	모든 값이 원자값	반복 그룹
2NF	부분 함수 종속 제거	복합키 일부에 종속
3NF	이행적 함수 종속 제거	A → B → C 종속
BCNF	모든 결정자가 후보키	비후보키 결정자

실무에서는 어디까지?

많은 서비스에서는 **3NF까지** 적용해도 충분한 경우가 많습니다. BCNF까지 분해하면 테이블 수가 많아지고 `JOIN`이 늘어나면서 조회 쿼리가 복잡해질 수 있습니다.

오히려 실무에서는 의도적으로 정규화를 풀어 **반정규화(Denormalization)** 하는 경우도 많습니다.

- **읽기 성능이 중요한 조회 테이블** — `JOIN`을 줄이기 위해 중복 허용
- **집계/리포팅 테이블** — 매번 `GROUP BY`하는 대신 미리 합산값을 저장
- **캐시 성격의 컬럼** — 댓글 수를 게시글 테이블에 함께 저장

핵심은 **정규화 원칙을 이해한 상태에서 트레이드오프를 판단**하는 것입니다.

정규화의 장점과 단점

장점

- **데이터 무결성 보장** — 중복이 사라지므로 갱신 시 불일치가 발생하지 않습니다
- **저장 공간 절약** — 같은 데이터를 여러 행에 반복 저장하지 않아 디스크를 효율적으로 사용합니다
- **유지보수 용이** — 데이터 변경 지점이 한 곳이므로 수정 범위가 명확합니다
- **확장성** — 새로운 요구사항이 생겨도 기존 테이블 구조를 크게 바꾸지 않고 테이블을 추가할 수 있습니다

단점

- **조회 복잡도 증가** — 테이블이 분리될수록 `JOIN`이 늘어나 읽기 쿼리가 복잡해지거나 느려질 수 있습니다
- **설계 복잡도 증가** — 테이블 수가 많아지면 ERD가 복잡해지고 쿼리 작성이 어려워집니다
- **쓰기 성능 오버헤드** — 하나의 트랜잭션에서 여러 테이블을 동시에 갱신해야 하는 경우가 생깁니다
- **과도한 정규화의 함정** — 읽기가 압도적으로 많은 서비스에서 BCNF까지 고집하면 오히려 전체 성능이 떨어집니다

> **참고:** 정규화와 반정규화는 양자택일이 아닙니다. **쓰기 정합성이 중요한 테이블은 정규화, 읽기 속도가 중요한 테이블은 반정규화**로 목적에 맞게 섞어 쓰는 것이 실무의 일반적인 접근입니다.

정리

1. **1NF** — 하나의 셀에 하나의 값만 저장합니다
2. **2NF** — 복합키의 일부에만 종속되는 컬럼을 분리합니다
3. **3NF** — 비키 컬럼이 다른 비키 컬럼을 결정하지 않도록 분리합니다
4. **BCNF** — 모든 결정자가 후보키인지 확인합니다
5. **실무 판단** — 정규화를 알아야 반정규화도 올바르게 할 수 있습니다

PART 2

2부. 인덱스와 실행 계획

Chapter 2

인덱스 완전 정복 — B+Tree 구조부터 실행 계획 읽기까지

#인덱스 #실행 계획

인덱스가 왜 빠른지, 복합 인덱스 컬럼 순서가 왜 중요한지, EXPLAIN은 어떻게 읽는지 예제와 함께 정리합니다.

인덱스, 왜 필요한가요?

테이블에 100만 행이 있다고 가정합니다. `WHERE name = '김철수'`를 인덱스 없이 실행하면 어떻게 될까요?

- DB는 **첫 번째 행부터 마지막 행까지** 하나씩 비교합니다
- 이것을 **풀 테이블 스캔(Full Table Scan)** 이라고 합니다
- 데이터가 늘어날수록 조회 시간이 **선형으로 증가**합니다

인덱스는 책의 목차와 같습니다. 목차 없이 원하는 내용을 찾으려면 첫 페이지부터 넘겨야 하지만, 목차가 있으면 해당 페이지로 바로 이동할 수 있습니다. DB 인덱스도 마찬가지로, 원하는 데이터의 **위치를 빠르게 찾아주는 자료구조**입니다.

> **기준:** 이 글은 MySQL 8.x와 InnoDB를 기준으로 설명합니다. 다른 DBMS나 스토리지 엔진에서는 인덱스 구조와 실행 계획 표기가 다를 수 있습니다.

Phase 1. B+Tree 인덱스의 구조

왜 하필 B+Tree인가요?

MySQL InnoDB는 기본 인덱스를 흔히 **B+Tree 계열 구조**로 설명합니다. 여기서는 InnoDB 관점에서 B+Tree를 중심으로 설명하겠습니다.

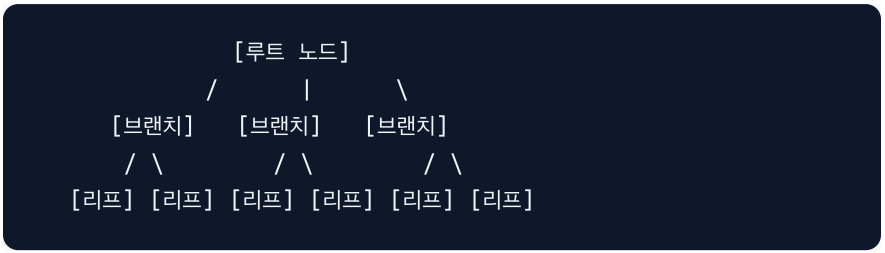
- **B-Tree** — 브랜치 노드에도 데이터를 저장합니다. 운이 좋으면 리프까지 내려가지 않고 중간에서 데이터를 찾을 수 있습니다
- **B+Tree** — 데이터는 **리프 노드에만** 저장하고, 리프 노드끼리 **링크드 리스트**로 연결합니다. 브랜치 노드에는 키만 저장하므로 하나의 노드에 더 많은 키를 담을 수 있어 트리 높이가 낮아집니다

DB가 B+Tree를 선택한 이유는 명확합니다.

- **항상 균형을 유지** — 어떤 데이터를 찾든 트리의 높이가 일정합니다
- **범위 검색에 유리** — 리프 노드가 정렬된 상태로 양방향 링크드 리스트로 연결되어 있어 `BETWEEN`, `>`, `<` 같은 범위 쿼리에 효율적입니다
- **디스크 I/O 최소화** — 하나의 노드에 여러 키를 저장해서 트리의 높이를 낮게 유지합니다

B+Tree의 구조

B+Tree는 세 가지 노드로 구성됩니다.



- **루트 노드(Root Node)** — 트리의 시작점입니다. 검색은 항상 여기서 시작합니다
- **브랜치 노드(Branch Node)** — 중간 경로입니다. 어느 방향으로 내려갈지 안내하는 이정표 역할을 합니다
- **리프 노드(Leaf Node)** — 실제 인덱스 키 값과 데이터의 위치(또는 InnoDB에서는 PK 참조값)가 저장됩니다. InnoDB 기준으로 리프 노드가 연결되어 있어 정렬 순서대로 범위 탐색할 수 있습니다

검색 과정 예시

`age` 컬럼에 인덱스가 있고, `WHERE age = 28`을 실행한다고 가정합니다.

```
1. 루트 노드에서 28이 어느 범위에 속하는지 확인 → 왼쪽 브랜치로 이동
2. 브랜치 노드에서 다시 범위 확인 → 해당 리프 노드로 이동
3. 리프 노드에서 28을 찾고, 저장된 포인터로 실제 데이터 행에 접근
```

트리의 높이가 3이라면, **단 3번의 노드 접근**으로 원하는 데이터를 찾을 수 있습니다. 100만 행이든 1,000만 행이든 B+Tree의 높이는 보통 3~4 수준이므로, 데이터 양이 늘어도 검색 속도가 크게 변하지 않습니다.

> **참고:** B+Tree의 하나의 노드는 수백 개의 키를 저장할 수 있어 분기 계수(branching factor)가 매우 높습니다. 덕분에 1,000만 행이라도 트리 높이가 3~4에 불과하며, 디스크 I/O 횟수도 그만큼만 발생합니다. 풀 테이블 스캔의 O(N)과는 비교할 수 없는 차이입니다.

Phase 2. 클러스터드 vs 논클러스터드 인덱스

클러스터드 인덱스 (Clustered Index)

데이터 자체가 인덱스 순서대로 물리적으로 정렬되어 저장되는 인덱스입니다.

- 테이블당 **하나만** 존재할 수 있습니다 (물리적 정렬은 하나의 기준만 가능)
- MySQL(InnoDB)에서는 **기본키(PK)가 곧 클러스터드 인덱스**입니다
- 리프 노드에 **실제 데이터 행 전체**가 저장됩니다

```
-- 클러스터드 인덱스 (PK = id)
리프 노드: [id=1, name='김철수', age=28] → [id=2, name='이영희',
age=25] → ...
```

논클러스터드 인덱스 (Non-Clustered Index, 보조 인덱스)

별도의 공간에 인덱스 키와 PK 값을 저장하는 인덱스입니다.

- 테이블에 **여러 개** 만들 수 있습니다
- 리프 노드에는 인덱스 키 값과 **PK 값(포인터)** 이 저장됩니다
- 데이터를 가져오려면 PK로 **다시 한번 클러스터드 인덱스를 탐색**해야 합니다

```
-- 논클러스터드 인덱스 (name)
리프 노드: ['김철수', PK=1] → ['이영희', PK=2] → ...
          ↓                    ↓
    클러스터드 인덱스에서      클러스터드 인덱스에서
    PK=1인 행 조회             PK=2인 행 조회
```

이 두 번째 조회를 **랜덤 I/O**라고 하며, 조회할 행이 많아지면 오히려 풀 테이블 스캔보다 느려질 수 있습니다. 이것이 옵티마이저가 대량 데이터 조회 시 인덱스를 사용하지 않는 이유입니다.

> **참고:** MySQL(InnoDB)에서 보조 인덱스의 리프 노드에 저장되는 것은 행의 물리적 주소가 아니라 **PK 값**입니다. 따라서 PK가 크면(예: UUID) 모든 보조 인덱스의 크기도 함께 커집니다. PK를 `BIGINT AUTO_INCREMENT`로 설정하는 것이 유리한 이유 중 하나입니다.

Phase 3. 복합 인덱스와 컬럼 순서

복합 인덱스란?

두 개 이상의 컬럼을 조합해서 만드는 인덱스입니다.

```
CREATE INDEX idx_name_age ON users (name, age);
```

컬럼 순서가 중요한 이유

복합 인덱스는 **왼쪽 컬럼부터 순서대로** 정렬됩니다. `(name, age)` 인덱스는 이렇게 정렬됩니다.

```
김철수, 25
김철수, 28
박민수, 27
이영희, 22
이영희, 30
```

먼저 `name`으로 정렬하고, 같은 `name` 안에서 `age`로 정렬합니다. 이 순서 때문에 다음과 같은 차이가 생깁니다.

```
-- ✅ 인덱스 사용 가능 (왼쪽부터 순서대로 사용)
WHERE name = '김철수'
WHERE name = '김철수' AND age = 28

-- ❌ 일반적인 복합 인덱스 탐색에는 부적합 (첫 번째 컬럼을 건너뜀)
WHERE age = 28
```

`age`만으로 검색하면 일반적인 leftmost prefix 탐색에는 맞지 않습니다. `name`이 정해지지 않은 상태에서 `age` 기준 정렬을 바로 활용하기 어렵기 때문입니다. 이것을 **최좌측 접두사 규칙(Leftmost Prefix Rule)** 이라고 합니다. 다만 MySQL 8.x의 skip scan 같은 예외 최적화가 적용되는 경우는 있을 수 있습니다.

어떤 컬럼을 앞에 놓아야 할까?

일반적인 원칙은 다음과 같습니다.

1. **동등 조건(`=`)으로 자주 사용하는 컬럼**을 앞에 배치합니다
2. **범위 조건(`>`, `<`, `BETWEEN`)으로 사용하는 컬럼**은 뒤에 배치합니다
3. **카디널리티(고유 값의 수)가 높은 컬럼**을 앞에 두면 검색 범위가 빠르게 좁혀집니다

```
-- status는 값이 3~4종류, created_at은 범위 검색
-- → status를 앞에, created_at을 뒤에
CREATE INDEX idx_status_created ON orders (status,
created_at);

-- 이렇게 하면 아래 쿼리가 효율적으로 동작합니다
WHERE status = 'PAID' AND created_at > '2026-01-01'
```

> **참고:** 범위 조건이 사용된 컬럼 **이후의 컬럼**은 일반적인 B+Tree 탐색에서 스캔 범위를 더 좁히기 어려워집니다. `(a, b, c)` 인덱스에서 `WHERE a = 1 AND b > 10 AND c = 100` 이라면, 보통 `a` 와 `b` 까지만 탐색 범위를 줄이고 `c` 는 추가 필터링으로 활용될 가능성이 큽니다.

Phase 4. 커버링 인덱스

커버링 인덱스란?

쿼리가 요청하는 **모든 컬럼이 인덱스에 포함**되어 있어서, 실제 데이터 행을 읽지 않고 **인덱스만으로 결과를 반환**할 수 있는 상태를 말합니다.

```
-- (name, age) 인덱스가 있을 때
SELECT name, age FROM users WHERE name = '김철수';
```

이 쿼리는 `name` 과 `age` 만 필요한데, 둘 다 인덱스에 있습니다. 따라서 클러스터드 인덱스(데이터 페이지)를 추가로 조회할 필요가 없습니다.

왜 빠른가요?

- **랜덤 I/O 제거** — 보조 인덱스 → 클러스터드 인덱스로 이동하는 과정이 사라집니다
- **읽는 데이터 양 감소** — 인덱스 페이지는 데이터 페이지보다 훨씬 작습니다 (필요한 컬럼만 저장)

EXPLAIN으로 확인하기

커버링 인덱스가 적용되면 `EXPLAIN` 결과의 `Extra` 컬럼에 **`Using index`** 가 표시됩니다.

```
EXPLAIN SELECT name, age FROM users WHERE name = '김철수';
```

```
+----+-------+------+-----------+
| id | type  | key  | Extra     |
+----+-------+------+-----------+
|  1 | ref   | idx  | Using index |
+----+-------+------+-----------+
```

> **참고:** 커버링 인덱스를 위해 `SELECT *` 대신 **필요한 컬럼만 명시**하는 습관이 중요합니다. `SELECT *`은 인덱스에 포함되지 않은 컬럼까지 요청하므로 커버링 인덱스를 활용할 수 없습니다.

Phase 5. EXPLAIN 실행 계획 읽기

EXPLAIN이란?

쿼리를 실행하기 전에 **옵티마이저가 어떤 계획으로 실행할지** 미리 보여주는 명령입니다.

```
EXPLAIN SELECT * FROM orders WHERE user_id = 1 AND status
= 'PAID';
```

핵심 컬럼 해석

```
+----+-------------+--------+------+---------------+------
+---------+------+------+-------------+
| id | select_type | table  | type | possible_keys | key
| key_len | ref  | rows | Extra       |
+----+-------------+--------+------+---------------+------
+---------+------+------+-------------+
```

가장 중요한 컬럼들을 하나씩 살펴보겠습니다.

type — 접근 방식

성능에 가장 큰 영향을 주는 컬럼입니다. **아래에서 위로 갈수록 일반적으로 더 좋은 접근 방식입니다.**

type	의미	성능
ALL	풀 테이블 스캔	최악
index	풀 인덱스 스캔 (인덱스 전체를 순회)	나쁨
range	인덱스 범위 스캔 (BETWEEN, >, <)	보통
ref	인덱스를 사용한 동등 비교 (비유니크)	좋음
eq_ref	인덱스를 사용한 동등 비교 (유니크/PK)	매우 좋음
const / system	PK 또는 유니크 인덱스로 1행 조회	최고

key — 실제 사용된 인덱스

`possible_keys`는 사용 **가능한** 인덱스 후보이고, `key`는 옵티마이저가 **실제로 선택한** 인덱스입니다. `key`가 `NULL`이면 인덱스를 사용하지 않았다는 뜻입니다.

rows — 예상 조회 행 수

옵티마이저가 **얼마나 많은 행을 읽어야 하는지** 추정한 값입니다. 실제 값과 다를 수 있지만, 이 숫자가 크면 쿼리가 비효율적일 가능성이 높습니다.

Extra — 추가 정보

자주 보이는 값들입니다.

Extra	의미
Using index	커버링 인덱스 사용 (데이터 페이지 접근 없음)
Using where	WHERE 조건으로 추가 필터링
Using filesort	인덱스 순서와 다른 정렬이 필요해 별도 정렬 수행
Using temporary	임시 테이블 생성 (GROUP BY, DISTINCT 등)

`Using filesort`와 `Using temporary`가 함께 나타나면 성능 개선이 필요한 신호입니다.

> **참고:** `EXPLAIN ANALYZE` (MySQL 8.0.18+)를 사용하면 예상 값이 아닌 **실제 실행 결과**를 확인할 수 있습니다. 실행 계획과 실제 성능 차이가 의심될 때 유용합니다.

Phase 6. 인덱스가 안 타는 케이스

인덱스를 만들어 놓고도 사용되지 않는 대표적인 경우들입니다.

인덱스 컬럼에 가공을 하는 경우

```
-- ❌ 함수 적용 → 인덱스 무시
WHERE YEAR(created_at) = 2026

-- ✅ 범위 조건으로 변환
WHERE created_at ≥ '2026-01-01' AND created_at < '2027-
01-01'
```

인덱스는 **원본 값 기준**으로 정렬되어 있습니다. 함수를 적용하면 정렬 순서가 달라질 수 있어 B+Tree를 활용할 수 없습니다.

묵시적 타입 변환

```
-- user_id 컬럼이 VARCHAR인데 숫자로 비교
-- ❌ 비교 과정에서 컬럼 값을 숫자로 해석하게 되어 비효율적일 수 있음
WHERE user_id = 123

-- ✅ 문자열로 비교
WHERE user_id = '123'
```

MySQL에서 문자열 컬럼을 숫자 리터럴과 비교하면, 문자열 정렬 기준으로 만들어 둔 인덱스를 그대로 활용하기 어려워질 수 있습니다. 가장 안전한 방법은 **컬럼 타입과 비교값 타입을 맞추는 것**입니다.

OR 조건

```
-- ❌ OR로 다른 컬럼을 연결하면 인덱스 활용이 어려움
WHERE name = '김철수' OR age = 28

-- ✅ 각각 인덱스가 있다면 UNION으로 분리
SELECT * FROM users WHERE name = '김철수'
UNION ALL
SELECT * FROM users WHERE age = 28 AND name ≠ '김철수'
```

NOT, != 부정 조건

```
-- ❌ 부정 조건은 옵티마이저가 비효율적이라 판단하여 인덱스를 사용하지 않는 경우가 많음
WHERE status ≠ 'DELETED'

-- ✅ 긍정 조건으로 변환 가능하면 변환
WHERE status IN ('ACTIVE', 'PENDING', 'PAID')
```

LIKE의 와일드카드 위치

```
-- ❌ 앞에 % → 인덱스 무시
WHERE name LIKE '%철수'

-- ✅ 뒤에 % → 인덱스 사용 가능
WHERE name LIKE '김%'
```

`%`가 앞에 오면 시작 지점을 특정할 수 없으므로 B+Tree를 탐색할 수 없습니다.

조회 대상이 너무 많은 경우

인덱스로 조회할 행이 전체 데이터의 상당 부분을 차지하면, 옵티마이저는 인덱스를 거치는 것보다 풀 테이블 스캔이 더 효율적이라고 판단합니다. 정확한 임계값은 데이터 분포, 페이지 크기 등에 따라 다르지만, 경험적으로 **약 10~25% 이상**이면 풀 스캔이 선택되는 경우가 많습니다. 보조 인덱스의 랜덤 I/O 비용이 순차 스캔보다 커지기 때문입니다.

> **참고:** 인덱스를 만들었는데 `EXPLAIN`에서 사용되지 않는다면 위 케이스 중 하나에 해당하는 경우가 대부분입니다. `EXPLAIN`으로 확인하는 습관이 가장 확실한 예방법입니다.

한눈에 보는 인덱스 핵심 정리

개념	핵심 포인트
B+Tree	균형 트리, O(log N) 탐색, 범위 검색에 유리
클러스터드 인덱스	데이터 자체가 PK 순서로 정렬, 테이블당 1개
논클러스터드 인덱스	별도 저장, PK로 다시 조회 (랜덤 I/O)
복합 인덱스	최좌측 접두사 규칙, 동등 조건 → 범위 조건 순서로 배치
커버링 인덱스	인덱스만으로 결과 반환, SELECT * 지양
EXPLAIN	type, key, rows, Extra를 확인

정리

1. **B+Tree** — 대부분의 RDBMS가 사용하는 기본 인덱스 구조이며, O(log N)으로 빠른 검색을 보장합니다
2. **클러스터드 vs 논클러스터드** — PK는 곧 클러스터드 인덱스이며, 보조 인덱스는 PK를 통해 데이터에 접근합니다
3. **복합 인덱스** — 컬럼 순서가 성능을 결정합니다. 동등 조건을 앞에, 범위 조건을 뒤에 배치합니다
4. **커버링 인덱스** — 인덱스만으로 쿼리를 처리하면 랜덤 I/O를 제거할 수 있습니다

5. **EXPLAIN** — 쿼리 튜닝의 기본 출발점입니다. 인덱스를 만들었으면 실행 계획으로 확인하는 습관이 중요합니다
6. **인덱스 함정** — 컬럼 가공, 타입 불일치, 앞쪽 와일드카드 등은 인덱스를 무력화합니다

Chapter 3

인덱스가 안 타는 이유 — 만들어도 느린 쿼리의 공통 원인

#인덱스 #쿼리 최적화

인덱스가 있는데도 풀 스캔이 나오는 이유를 함수 가공, 암묵적 형변환, 복합 인덱스 순서, 선택도 관점에서 정리합니다.

인덱스가 안 타는 이유, 왜 알아야 하나요?

인덱스 기본 글에서 인덱스는 데이터를 빠르게 찾기 위한 구조라고 정리했습니다. 그런데 실무에서는 인덱스를 만들어도 이런 상황이 자주 나옵니다.

- 분명 인덱스를 만들었는데 `EXPLAIN` 결과가 `ALL` 입니다
- `WHERE` 조건이 있는데도 풀 테이블 스캔이 나옵니다
- 인덱스를 추가했는데 쿼리가 거의 빨라지지 않습니다

이때 중요한 것은 "인덱스가 있다"가 아니라 **옵티마이저가 그 인덱스를 쓰는 편이 유리하다고 판단하느냐** 입니다. 이 글에서는 인덱스가 안 타는 대표적인 이유를 원인과 판단 기준 중심으로 정리합니다.

> **기준:** 이 글은 MySQL 8.x와 InnoDB를 기준으로 설명합니다. 다른 DBMS나 스토리지 엔진에서는 실행 계획 표기와 세부 동작이 다를 수 있습니다.

Phase 1. 인덱스가 있어도 항상 쓰는 것은 아닙니다

인덱스가 있으면 DB가 무조건 그 인덱스를 사용할 것 같지만, 실제로는 그렇지 않습니다. 옵티마이저는 여러 실행 계획 중에서 **비용(cost)이 더 낮다고 추정되는 계획**을 선택합니다.

왜 풀 스캔을 선택할까?

```sql
EXPLAIN
SELECT *
FROM users
WHERE gender = 'F';
```

`gender` 컬럼에 인덱스가 있어도, 테이블의 절반 이상이 `F`라면 어떨까요? 인덱스를 타고 많은 행을 찾은 뒤 다시 테이블로 돌아가 읽는 비용이, 처음부터 순서대로 전부 읽는 비용보다 더 클 수 있습니다.

즉, 인덱스가 안 타는 이유는 종종 **인덱스를 못 써서**가 아니라 **써도 이득이 없어서**입니다.

핵심 판단 기준

옵티마이저는 대략 이런 요소를 함께 봅니다.

- 조건이 얼마나 많은 행을 걸러내는가
- 인덱스를 타고 찾은 뒤 테이블을 다시 읽어야 하는가
- 정렬과 필터링을 인덱스로 같이 해결할 수 있는가
- 통계 정보상 어느 계획이 더 저렴해 보이는가

이 네 가지를 이해하면 **왜 옵티마이저가 인덱스를 선택하지 않았는지**를 훨씬 쉽게 해석할 수 있습니다.

Phase 2. 인덱스 컬럼을 가공하면 B+Tree를 활용하기 어렵습니다

인덱스는 **원본 값 기준**으로 정렬되어 있습니다. 그래서 인덱스 컬럼에 함수를 적용하면 정렬 순서를 그대로 활용하기 어려워집니다.

함수 적용

```
-- ❌ created_at 인덱스가 있어도 비효율적
WHERE YEAR(created_at) = 2026

-- ✅ 원본 값 기준 범위 검색
WHERE created_at ≥ '2026-01-01'
  AND created_at < '2027-01-01'
```

`YEAR(created_at)`는 인덱스에 저장된 원본 값이 아니라 **가공된 결과**입니다. B+Tree는 `2026-03-10 12:34:56` 같은 원본 값을 기준으로 정렬되어 있으므로, 연도만 잘라 비교하면 처음부터 끝까지 계산해 봐야 할 수 있습니다.

문자열 조작

```
-- ❌ name 인덱스 활용이 어려움
WHERE LEFT(name, 3) = '김철수'

-- ✅ 전방 일치 검색
WHERE name LIKE '김철수%'

-- ❌ 앞에 와일드카드가 붙으면 시작 위치를 바로 찾기 어려움
WHERE name LIKE '%철수'
```

`LIKE '%철수'`처럼 앞에 와일드카드가 붙는 경우도 마찬가지입니다. B+Tree는 **왼쪽부터 정렬**되어 있으므로, 앞부분이 고정되지 않으면 시작 위치를 바로 찾기 어렵습니다.

> **참고:** MySQL 8.0.13+에서는 함수 기반 인덱스(functional index)를 사용할 수 있지만, 기본 원칙은 그대로입니다. 함수 결과를 따로 인덱싱하지 않는 이상 원본 컬럼 인덱스는 활용하기 어렵습니다.

Phase 3. 암묵적 형변환이 인덱스를 망칠 수 있습니다

SQL은 타입이 다르면 내부적으로 형변환을 시도합니다. 특히 MySQL에서는 **문자열 컬럼을 숫자 리터럴과 비교하는 경우**처럼 비교 과정에서 컬럼 값을 변환해야 하면, 인덱스를 비효율적으로 사용할 수 있습니다.

숫자 컬럼 vs 문자열 비교

```
-- user_id가 VARCHAR인데 숫자로 비교
WHERE user_id = 123
```

`user_id`가 `VARCHAR`라면 `WHERE user_id = 123`은 문자열을 원본 값 그대로 비교하는 것이 아니라, 값을 숫자로 해석하는 비교로 바뀔 수 있습니다. 이렇게 되면 문자열 정렬 기준으로 만들어 둔 인덱스를 그대로 활용하기 어려워질 수 있습니다.

반대로 숫자 컬럼을 `'123'`처럼 문자열 리터럴과 비교하는 경우는 안전하게 숫자로 변환되면 인덱스를 탈 수도 있지만, 타입을 섞는 습관 자체는 피하는 편이 좋습니다.

가장 안전한 방법은 **컬럼 타입과 비교값 타입을 맞추는 것**입니다.

```
-- ✅ 타입 일치
WHERE user_id = '123'
```

조인 조건의 타입 불일치

```
-- orders.user_id: BIGINT
-- users.id: VARCHAR
SELECT *
FROM orders o
JOIN users u ON o.user_id = u.id;
```

조인 키 타입이 다르면 한쪽 값을 계속 변환해야 하고, 그 결과 인덱스를 제대로 활용하지 못할 수 있습니다. 조인 성능이 이상하게 느린 경우에는 **양쪽 컬럼 타입부터** 먼저 확인하는 편이 좋습니다.

Phase 4. 복합 인덱스는 "있다"보다 "순서가 맞다"가 중요합니다

복합 인덱스는 컬럼을 여러 개 넣었다고 끝나지 않습니다. **어떤 순서로 정렬되어 있는가**가 핵심입니다.

왼쪽부터 맞아야 합니다

```
CREATE INDEX idx_status_created_at ON orders (status,
created_at);
```

이 인덱스가 있을 때:

```
-- ✅ 잘 맞는 경우
WHERE status = 'PAID'
WHERE status = 'PAID' AND created_at ⩾ '2026-01-01'

-- ❌ 일반적인 탐색에는 부적합
WHERE created_at ⩾ '2026-01-01'
```

인덱스는 `status` 부터 정렬되어 있으므로, 첫 번째 컬럼을 건너뛰고 `created_at` 만으로 탐색하는 것은 일반적으로 비효율적입니다.

범위 조건 이후는 활용이 약해집니다

```
CREATE INDEX idx_a_b_c ON sample (a, b, c);

-- a는 동등 조건, b는 범위 조건
WHERE a = 1
  AND b > 10
  AND c = 100
```

이 경우 보통 `a`, `b` 까지는 인덱스를 잘 활용할 수 있지만, **범위 조건이 들어간 뒤의 `c` 는 일반적인 B+Tree 탐색에서 스캔 범위를 더 좁히기 어렵고**, 남더라도 추가 필터링 단계로 활용되는 경우가 많습니다.

즉, 복합 인덱스는 "컬럼이 다 들어갔다"보다 **동등 조건 → 범위 조건 → 정렬 조건 순서에 얼마나 잘 맞는가**가 중요합니다.

Phase 5. 선택도가 낮으면 인덱스가 손해일 수 있습니다

선택도(selectivity)는 조건이 **얼마나 많은 데이터를 걸러내는가**를 뜻합니다. 선택도가 낮으면 인덱스가 있어도 큰 도움이 되지 않습니다.

이런 컬럼은 주의해야 합니다

- 성별: `M`, `F`
- 상태값: `ACTIVE`, `INACTIVE`
- 삭제 여부: `Y`, `N`
- boolean 플래그

```
CREATE INDEX idx_is_deleted ON posts (is_deleted);

SELECT *
FROM posts
WHERE is_deleted = 'N';
```

전체 데이터의 95%가 `N`이라면, 이 조건은 거의 아무것도 걸러내지 못합니다. 그러면 인덱스를 타고 대량의 PK를 읽은 뒤 테이블로 다시 접근하는 것보다, 풀 스캔이 더 낫다고 판단될 수 있습니다.

해결 방향

선택도가 낮은 단일 컬럼 인덱스는 단독으로 두기보다, **다른 조건과 결합한 복합 인덱스**로 설계하는 경우가 많습니다.

```
-- is_deleted 단독보다, 실제 조회 패턴에 맞춘 조합이 유리할 수 있음
CREATE INDEX idx_is_deleted_created_at ON posts
(is_deleted, created_at);
```

물론 이 경우에도 `is_deleted`가 항상 먼저 와야 하는지는 쿼리 패턴에 따라 달라집니다. 핵심은 **실제 WHERE 조건 조합**을 기준으로 인덱스를 설계해야 한다는 점입니다.

Phase 6. 조회 행이 너무 많으면 인덱스보다 풀 스캔이 낫습니다

InnoDB의 보조 인덱스는 보통 "인덱스 탐색 → PK 조회 → 실제 행 읽기" 과정을 거칩니다. 그런데 조건에 맞는 행이 너무 많으면 이 랜덤 I/O가 크게 늘어납니다.

`SELECT *`가 특히 불리한 이유

```
SELECT *
FROM orders
WHERE status = 'PAID';
```

인덱스를 타더라도 결국 필요한 컬럼이 많으면 테이블 페이지를 다시 많이 읽어야 합니다. 조회 비율이 높을수록 보조 인덱스의 장점이 줄어듭니다.

반면 필요한 컬럼이 적고, 인덱스 안에서 해결되는 쿼리는 훨씬 유리합니다.

```
-- 커버링 인덱스 가능성
SELECT status, created_at
FROM orders
WHERE status = 'PAID';
```

즉, 같은 조건이어도 `SELECT *`인지, 필요한 컬럼만 조회하는지에 따라 옵티마이저 판단이 달라질 수 있습니다.

Phase 7. 정렬과 조건이 따로 놀면 인덱스 이점을 잃습니다

인덱스는 검색뿐 아니라 정렬에도 도움이 됩니다. 하지만 `WHERE`와 `ORDER BY`가 인덱스 순서와 맞지 않으면 별도 정렬이 필요해집니다.

정렬 컬럼 순서가 맞지 않는 경우

```
CREATE INDEX idx_status_created_at ON orders (status,
created_at);

SELECT *
FROM orders
WHERE status = 'PAID'
ORDER BY updated_at DESC;
```

이 인덱스는 `status`, `created_at` 기준 정렬만 자연스럽게 지원합니다. 그런데 `updated_at`으로 정렬하면, 필터링은 인덱스를 일부 활용하더라도 결국 `Using filesort`가 발생할 수 있습니다. 여기서 `filesort`는 "추가 정렬 단계"를 뜻하며, 항상 디스크 정렬을 의미하는 것은 아닙니다.

정렬 방향과 컬럼 구성이 함께 중요합니다

```
SELECT *
FROM orders
WHERE user_id = 42
ORDER BY created_at DESC
LIMIT 20;
```

이 쿼리가 자주 호출된다면, `WHERE`에서 먼저 `user_id`를 좁힌 뒤 `created_at` 순서로 정렬할 수 있는 `user_id, created_at` 인덱스는 매우 강력할 수 있습니다. 반대로 인덱스가 `created_at, user_id` 순서면 이 쿼리 패턴에는 기대만큼 효율적이지 않을 수 있습니다.

즉, 인덱스는 `WHERE`만 보는 것이 아니라 **필터링 + 정렬 + LIMIT**까지 한 세트로 봐야 합니다.

Phase 8. 통계 정보가 틀리면 실행 계획도 틀릴 수 있습니다

옵티마이저는 실제 데이터를 전부 읽어 보고 판단하지 않습니다. **통계 정보(statistics)**를 기반으로 어느 계획이 저렴할지 추정합니다.

왜 통계가 중요할까?

- 데이터 분포가 최근에 크게 바뀌었는데 통계가 오래됨
- 특정 값에 데이터가 몰려 있는데 샘플링이 부정확함
- 개발 환경과 운영 환경의 데이터 분포가 완전히 다름

이런 경우 실제로는 인덱스가 유리한데도, 옵티마이저가 풀 스캔을 고를 수 있습니다.

그래서 무엇을 확인해야 할까?

- `EXPLAIN`의 `rows` 추정치가 비정상적으로 큰지
- 최근 데이터 분포가 크게 바뀌지 않았는지
- 통계 갱신이 필요한 상황인지

MySQL에서는 `ANALYZE TABLE`로 통계를 갱신할 수 있습니다.

```
ANALYZE TABLE orders;
```

> **참고:** 통계가 문제인 경우는 "쿼리 문법이 틀렸다"가 아니라 "옵티마이저가 잘못 추정했다"에 가깝습니다. 그래서 같은 쿼리가 어떤 환경에서는 빠르고, 어떤 환경에서는 느릴 수 있습니다.

한눈에 보는 대표 원인

실무에서는 인덱스가 안 탈 때 아래 항목을 순서대로 의심해 보면 원인을 빠르게 좁힐 수 있습니다.

원인	왜 인덱스가 안 타는가	대표 대응
컬럼 가공	원본 값 정렬을 활용할 수 없음	범위 조건, 함수 기반 인덱스 검토
암묵적 형변환	컬럼/비교값 타입 불일치	타입 일치
복합 인덱스 순서 불일치	최좌측 접두사(leftmost prefix)를 못 맞춤	컬럼 순서 재설계
선택도 낮음	너무 많은 행을 읽음	복합 인덱스, 조건 재검토
조회 행 수 과다	랜덤 I/O가 풀 스캔보다 비쌈	커버링 인덱스, 조회 컬럼 축소
정렬 조건 불일치	필터링과 정렬을 같이 해결 못 함	`WHERE` + `ORDER BY` 기준 재설계
통계 부정확	옵티마이저 비용 추정 오류	통계 갱신, 데이터 분포 확인

정리

1. **인덱스가 있다고 항상 사용하는 것은 아닙니다** — 옵티마이저는 가장 싸 보이는 계획을 고릅니다

2. **컬럼 가공과 암묵적 형변환은 가장 흔한 실수입니다** — 원본 값 기준 탐색이 깨질 수 있습니다
3. **복합 인덱스는 순서가 핵심입니다** — 컬럼이 들어 있다는 사실보다 WHERE, ORDER BY 패턴에 맞는지가 중요합니다
4. **선택도가 낮거나 조회 행이 너무 많으면 풀 스캔이 더 나을 수 있습니다** — 인덱스도 비용이 드는 구조입니다
5. **정렬과 LIMIT까지 함께 봐야 합니다** — 좋은 인덱스는 검색과 정렬을 같이 해결합니다
6. **실행 계획은 통계 정보에 의존합니다** — 통계가 틀리면 옵티마이저 판단도 틀릴 수 있습니다

Chapter 4

실행 계획 완전 정복 — EXPLAIN으로 쿼리 옵티마이저의 선택을 읽는 법

#실행 계획 #쿼리 최적화

EXPLAIN 출력의 각 컬럼이 무엇을 뜻하는지, 옵티마이저가 왜 그 계획을 골랐는지, 조인과 서브쿼리는 어떻게 읽는지 예제와 함께 정리합니다.

실행 계획, 왜 읽을 줄 알아야 하나요?

인덱스 기본 글에서 `EXPLAIN`의 핵심 컬럼을 간단히 살펴봤고, 인덱스가 안 타는 이유에서 옵티마이저가 인덱스를 무시하는 원인을 정리했습니다. 하지만 실무에서는 이런 상황이 생깁니다.

- `EXPLAIN` 결과가 여러 행인데 어느 행부터 읽어야 할지 모르겠습니다
- `type`이 `ref`인데 왜 느린지 감이 안 잡힙니다
- 서브쿼리와 조인이 섞인 쿼리의 실행 순서를 알 수 없습니다
- `EXPLAIN`의 예상 `rows`와 실제 성능이 다른 것 같습니다

`EXPLAIN`은 단순히 "인덱스를 탔는가"를 확인하는 도구가 아닙니다. **옵티마이저가 왜 그런 계획을 세웠는지**를 이해해야 올바른 튜닝 방향을 잡을 수 있습니다. 이 글에서는 MySQL을 기준으로 설명합니다.

Phase 1. 옵티마이저는 어떻게 실행 계획을 결정하나요?

비용 기반 옵티마이저(CBO)

MySQL은 **비용 기반 옵티마이저(Cost-Based Optimizer)** 를 사용합니다. 쿼리를 실행하는 방법은 여러 가지가 있고, 옵티마이저는 각 방법의 **예상 비용(cost)** 을 계산한 뒤 가장 저렴한 계획을 선택합니다.

```
쿼리 입력
  ↓
파싱 → 문법 확인, 구문 트리 생성
  ↓
옵티마이저 → 가능한 실행 계획 나열 → 각 계획의 비용 추정 → 최저 비용 계획 선택
  ↓
실행 엔진 → 선택된 계획대로 데이터 접근
  ↓
결과 반환
```

비용을 추정하는 재료

옵티마이저가 비용을 계산할 때 사용하는 주요 정보입니다.

- **테이블 통계** — 테이블의 전체 행 수, 인덱스별 카디널리티(고유 값 수)
- **데이터 분포** — 특정 값이 몇 건이나 있는지에 대한 히스토그램(MySQL 8.0+)

- **I/O 비용** — 디스크에서 페이지를 읽는 비용과 메모리에서 읽는 비용의 차이
- **CPU 비용** — 행을 비교하고 정렬하는 연산 비용

통계가 부정확하면 옵티마이저의 판단도 틀릴 수 있습니다. 인덱스가 안 타는 이유에서 다뤘듯이, `ANALYZE TABLE`로 통계를 갱신하면 실행 계획이 바뀌는 경우가 종종 있습니다.

> **참고:** MySQL에서 옵티마이저가 계산한 비용을 직접 확인하려면 `EXPLAIN FORMAT=JSON`을 사용합니다. 결과에 포함된 `query_cost` 필드가 옵티마이저가 추정한 총 비용입니다.

Phase 2. EXPLAIN 출력 컬럼 전체 해석

`EXPLAIN`을 실행하면 다음과 같은 컬럼이 출력됩니다.

```
EXPLAIN
SELECT o.id, o.status, u.name
FROM orders o
JOIN users u ON o.user_id = u.id
WHERE o.status = 'PAID'
  AND u.grade = 'VIP';
```

```
+----+-------------+-------+------+---------------+-------
--+---------+-----------+------+-------------+
| id | select_type | table | type | possible_keys | key
| key_len | ref       | rows | Extra       |
+----+-------------+-------+------+---------------+-------
--+---------+-----------+------+-------------+
|  1 | SIMPLE      | o     | ref  | idx_status    |
idx_status | 62   | const     | 1200 | Using where |
|  1 | SIMPLE      | u     | eq_ref | PRIMARY     |
PRIMARY | 8      | db.o.user_id | 1  | Using where |
+----+-------------+-------+------+---------------+-------
--+---------+-----------+------+-------------+
```

각 컬럼의 의미를 하나씩 살펴보겠습니다.

`id` — 쿼리 블록 번호

같은 `id`를 가진 행은 **하나의 조인 블록** 안에서 실행됩니다. 위 예시처럼 `id`가 모두 1이면 한 번의 조인으로 처리된다는 뜻입니다.

`id`가 다르면 별도의 쿼리 블록입니다. 전통적인 `EXPLAIN`에서는 서브쿼리나 파생 테이블에서 `id`가 큰 쪽이 먼저 처리되는 것처럼 보이는 경우가 많지만, **항상 실행 순서를 보장하는 값은 아닙니다.**

```
id=1: 외부 쿼리
id=2: 서브쿼리 (먼저 실행)
```

`select_type` — 쿼리 블록의 종류

select_type	의미
SIMPLE	서브쿼리나 UNION 없는 단순 쿼리
PRIMARY	가장 바깥쪽 SELECT
SUBQUERY	서브쿼리의 첫 SELECT (스칼라, `IN`, `EXISTS` 등 포함)
DERIVED	FROM 절의 서브쿼리 (파생 테이블)
UNION	UNION의 두 번째 이후 SELECT
DEPENDENT SUBQUERY	외부 쿼리에 의존하는 서브쿼리 (상관 서브쿼리)

`DEPENDENT SUBQUERY`는 외부 쿼리의 값에 따라 **반복 평가될 수 있는** 서브쿼리라는 뜻이므로 성능에 주의해야 합니다.

`table` — 접근 대상 테이블

해당 행이 어떤 테이블을 읽는지 표시합니다. 별칭을 사용했다면 별칭이 표시됩니다. `<derived2>`처럼 보이면 `id=2`인 파생 테이블을 뜻합니다.

`type` — 접근 방식 (가장 중요한 컬럼)

옵티마이저가 테이블에서 행을 어떻게 찾는지를 나타냅니다. 성능에 가장 큰 영향을 주는 컬럼입니다.

type	의미	읽는 행 수
`system`	테이블에 행이 1개 (시스템 테이블)	1
`const`	PK 또는 유니크 인덱스로 **정확히 1행** 조회	1
`eq_ref`	조인에서 PK/유니크 인덱스로 **1행씩** 매칭	조인 대상 행당 1
`ref`	비유니크 인덱스로 동등 비교	조건에 맞는 행 수
`range`	인덱스 범위 스캔 (`BETWEEN` , `>` , `<` , `IN`)	범위 안의 행 수
`index`	**인덱스 전체**를 처음부터 끝까지 스캔	인덱스 전체
`ALL`	**테이블 전체**를 처음부터 끝까지 스캔	테이블 전체

위에서 아래로 갈수록 읽는 행 수가 많아지므로 일반적으로 성능이 나빠집니다. 단, `range` 까지는 대부분 수용할 수 있는 수준이고, `index` 와 `ALL` 이 나타나면 개선을 검토해야 합니다.

`ref` 인데도 느리다면 `rows` 값을 확인합니다. 인덱스를 타더라도 매칭되는 행이 수만 건이면 여전히 느릴 수 있습니다.

`possible_keys` / `key` — 인덱스 후보와 실제 선택

- `possible_keys` — 옵티마이저가 **사용을 고려한** 인덱스 목록
- `key` — 그중 **실제로 선택한** 인덱스

`possible_keys` 에 인덱스가 있는데 `key` 가 `NULL` 이면, 옵티마이저가 "그 인덱스를 쓰는 것보다 풀 스캔이 싸다"고 판단한 것입니다.

`key_len` — 사용된 인덱스 길이

복합 인덱스에서 **몇 번째 컬럼까지 실제로 사용됐는지** 파악하는 단서입니다.

```
-- utf8mb3 기준: (status VARCHAR(20), created_at DATETIME)
복합 인덱스
-- status만 사용: key_len = 62  (20×3 + 2)
-- status + created_at 사용: key_len = 67  (62 + 5)
```

`key_len`이 예상보다 짧으면 복합 인덱스의 뒤쪽 컬럼이 활용되지 않은 것입니다. 이전 글에서 다뤘던 "범위 조건 이후 컬럼은 활용이 약해진다"가 여기서 구체적으로 확인됩니다.

> **참고:** `key_len`은 문자셋과 NULL 허용 여부에 따라 달라집니다. 위 숫자는 예시일 뿐입니다. 예를 들어 `VARCHAR(20)`이 `utf8mb4`이면 최대 바이트는 `20×4 + 2(길이 저장)` = 82이며, `NOT NULL`이 아니면 1바이트가 추가됩니다.

`ref` — 인덱스와 비교되는 값

인덱스 컬럼과 무엇을 비교했는지 보여줍니다.

- `const` — 상수 값과 비교 (`WHERE status = 'PAID'`)
- `db.o.user_id` — 다른 테이블의 컬럼과 비교 (조인)
- `NULL` — 인덱스를 사용하지 않았거나 범위 스캔

`rows` — 예상 조회 행 수

옵티마이저가 **읽어야 한다고 추정한 행 수**입니다. 실제 값과 다를 수 있지만, 튜닝 전후를 비교하는 지표로 유용합니다.

조인에서는 각 테이블의 `rows`를 보면 대략적인 탐색량을 짐작할 수 있지만, **단순히 모두 곱한 값이 정확한 총 비용은 아닙니다.** 실제로는 각 단계에서 조건으로 얼마나 걸러지는지(`filtered`)까지 함께 봐야 합니다. 위 예시처럼 `eq_ref` 조인에서는 `1200 × 1` 수준으로 이해해도 무리가 없지만, 복잡한 조인에서는 단순 곱셈만으로 판단하면 안 됩니다.

`Extra` — 추가 실행 정보

`Extra` 컬럼은 옵티마이저가 어떤 **추가 작업**을 하는지 보여줍니다. 여기서 성능 문제의 단서를 찾는 경우가 많습니다.

Extra	의미	주의도
`Using index`	커버링 인덱스로 처리 (테이블 접근 없음)	좋음
`Using where`	스토리지 엔진에서 읽은 뒤 서버에서 추가 필터링	보통
`Using index condition`	인덱스 컨디션 푸시다운 (ICP) 적용	보통~좋음
`Using temporary`	중간 결과 저장을 위해 임시 테이블 생성	주의
`Using filesort`	인덱스 순서로 해결되지 않아 별도 정렬 수행	주의
`Using join buffer`	조인 시 버퍼를 사용한 배치 처리 (BNL/Hash Join)	주의

`Using filesort`는 이름과 달리 반드시 디스크를 사용하는 것은 아닙니다. 메모리에서 처리될 수도 있지만, **인덱스 정렬을 활용하지 못했다**는 점이 핵심입니다.

`Using temporary` + `Using filesort`가 함께 나타나면 `GROUP BY`, `ORDER BY`, `DISTINCT` 처리 과정에서 임시 테이블을 만들고 다시 정렬하는 것이므로, 행 수가 많을 때 병목이 됩니다.

Phase 3. 조인 쿼리의 실행 계획 읽기

조인 실행 순서

MySQL에서 같은 `id`를 가진 행은 **위에서 아래로** 읽습니다. 위쪽 테이블이 **드라이빙 테이블(먼저 읽는 테이블)**, 아래쪽이 **드리븐 테이블(나중에 읽는 테이블)** 입니다.

```
+----+-------+------+---------+------+
| id | table | type | key     | rows |
+----+-------+------+---------+------+
|  1 | o     | ref  | idx_status | 1200 |  ← 드라이빙: orders를 먼저 읽음
|  1 | u     | eq_ref | PRIMARY |   1 |  ← 드리븐: 각 order에 대해 user를 PK로 조회
+----+-------+------+---------+------+
```

실행 흐름은 이렇습니다.

```
1. orders에서 status = 'PAID'인 행을 인덱스로 찾음 (약 1,200행)
2. 찾은 각 행의 user_id로 users 테이블을 PK 조회 (행당 1회)
3. 총 탐색량: 1,200 × 1 = 1,200
```

드라이빙 테이블 선택이 중요한 이유

옵티마이저는 **결과 행이 적은 테이블을 드라이빙으로** 선택하는 경향이 있습니다. 드라이빙 테이블에서 나온 행 수만큼 드리븐 테이블을 반복 조회하기 때문입니다.

```
드라이빙 1,200행 × 드리븐 1행 = 1,200회 접근
드라이빙 50,000행 × 드리븐 1행 = 50,000회 접근
```

같은 조인이라도 드라이빙 테이블이 바뀌면 성능이 크게 달라질 수 있습니다. `EXPLAIN`에서 첫 번째 테이블의 `rows`가 비정상적으로 크다면, 조건을 추가하거나 인덱스를 개선해서 드라이빙 테이블의 결과를 줄이는 것이 효과적입니다.

3개 이상 테이블 조인

```sql
EXPLAIN
SELECT o.id, u.name, p.product_name
FROM orders o
JOIN users u ON o.user_id = u.id
JOIN products p ON o.product_id = p.id
WHERE o.status = 'PAID';
```

```
+----+-------+--------+---------+------+
| id | table | type   | key     | rows |
+----+-------+--------+---------+------+
|  1 | o     | ref    | idx_status | 1200 |
|  1 | u     | eq_ref | PRIMARY |    1 |
|  1 | p     | eq_ref | PRIMARY |    1 |
+----+-------+--------+---------+------+
```

위에서 아래로 읽으면 됩니다. `orders` → `users` → `products` 순서로 접근하며, 총 탐색량은 `1,200 × 1 × 1 = 1,200`입니다. 드리븐 테이블이

모두 `eq_ref`이므로 조인 효율이 좋은 상태입니다.

만약 드리븐 테이블 중 하나가 `ALL`이라면 그 테이블에 적절한 인덱스가 없다는 뜻이므로, `1,200 × 전체 행 수`만큼 탐색이 발생할 수 있습니다.

Phase 4. 서브쿼리와 파생 테이블의 실행 계획

스칼라 서브쿼리

```
EXPLAIN
SELECT o.id,
       (SELECT u.name FROM users u WHERE u.id = o.user_id)
AS user_name
FROM orders o
WHERE o.status = 'PAID';
```

```
+----+--------------------+-------+--------+---------+----
--+
| id | select_type        | table | type   | key     |
rows |
+----+--------------------+-------+--------+---------+----
--+
|  1 | PRIMARY            | o     | ref    | idx_status |
1200 |
|  2 | DEPENDENT SUBQUERY | u     | eq_ref | PRIMARY |
1 |
+----+--------------------+-------+--------+---------+----
--+
```

`DEPENDENT SUBQUERY`는 외부 쿼리(`orders`)의 값에 따라 서브쿼리가 **반복 평가될 수 있다**는 의미입니다. 이 예시에서는 외부 행 수가 많아질수록 서브쿼리 평가 횟수도 함께 늘어날 가능성이 큽니다.

이 예시에서는 PK 조회(`eq_ref`)라 외부 행 수가 어느 정도 많아도 버틸 수 있지만, 서브쿼리 안에서 풀 스캔이 발생하면 외부 결과 수에 비례해 비용이 급격히 커질 수 있습니다.

FROM 절 서브쿼리 (파생 테이블)

```
EXPLAIN
SELECT d.status, d.cnt
FROM (
  SELECT status, COUNT(*) AS cnt
  FROM orders
  GROUP BY status
) d
WHERE d.cnt > 100;
```

```
+----+-------------+------------+-------+---------+------
+----------------+
| id | select_type | table      | type  | key     | rows |
Extra          |
+----+-------------+------------+-------+---------+------
+----------------+
|  1 | PRIMARY     | <derived2> | ALL   | NULL    |    5 |
Using where    |
|  2 | DERIVED     | orders     | index | idx_status |
50000 | Using index |
+----+-------------+------------+-------+---------+------
+----------------+
```

이 예시에서는 `id=2`의 파생 테이블이 먼저 계산된 뒤, 그 결과를 `id=1`에서 `<derived2>`로 읽는 형태로 해석할 수 있습니다. 즉 `orders` 테이블을 인덱스 전체 스캔하여 `GROUP BY`를 처리하고, 그 결과가 임시 테이블로 만들어진 다음 바깥 쿼리에서 사용됩니다.

파생 테이블은 **실체화(Materialization)** 되어 임시 테이블에 저장되므로, 결과가 클 경우 메모리나 디스크를 사용합니다. MySQL 8.0에서는 옵티마이저가 파생 테이블을 외부 쿼리에 **머지(Merge)** 하여 임시 테이블 생성을 피하기도 합니다.

> **참고:** `EXPLAIN`에서 `select_type`이 `DERIVED`인데 실제로는 머지되어 사라지는 경우, MySQL 8.0에서는 해당 행 자체가 출력되지 않을 수 있습니다. 실행 계획에 파생 테이블이 보이지 않는다면 머지가 적용된 것입니다.

Phase 5. EXPLAIN ANALYZE — 예상과 실제의 차이 확인

EXPLAIN은 예상, EXPLAIN ANALYZE는 실측

`EXPLAIN`의 `rows`는 **추정치**입니다. 통계 기반이므로 실제와 차이가 날 수 있습니다. `EXPLAIN ANALYZE`는 **쿼리를 실제로 실행**하고 각 단계의 실측 데이터를 보여줍니다.

```
-- MySQL 8.0.18+
EXPLAIN ANALYZE
SELECT o.id, u.name
FROM orders o
JOIN users u ON o.user_id = u.id
WHERE o.status = 'PAID';
```

```
→ Nested loop inner join  (cost=2145 rows=1200)
     (actual time=0.15..8.32 rows=1180 loops=1)
    → Index lookup on o using idx_status (status='PAID')
         (cost=540 rows=1200)
         (actual time=0.08..2.15 rows=1180 loops=1)
    → Single-row index lookup on u using PRIMARY
(id=o.user_id)
         (cost=1.00 rows=1)
         (actual time=0.004..0.004 rows=1 loops=1180)
```

출력 읽는 법

각 노드에 두 줄이 있습니다.

- **첫 줄 (cost, rows)** — 옵티마이저의 **예상** 비용과 행 수
- **둘째 줄 (actual time, rows, loops)** — **실제** 실행 시간, 반환 행 수, 반복 횟수

핵심 지표는 세 가지입니다.

지표	의미	확인 포인트
`actual time=A..B`	첫 행 반환까지 A ms, 마지막 행까지 B ms	B가 크면 해당 단계가 병목
`rows` (actual)	실제 반환된 행 수	예상 `rows` 와 크게 다르면 통계 부정확
`loops`	해당 단계가 반복 실행된 횟수	`loops` 가 크면 총 비용 = time × loops

위 예시에서 `rows=1200` (예상)과 `rows=1180` (실제)이 비슷하므로 통계가 정확한 상태입니다. 만약 예상은 100인데 실제가 50,000이라면 통계 갱신이 필요합니다.

> **참고:** `EXPLAIN ANALYZE`는 쿼리를 **실제로 실행**합니다. 따라서 지원되는 문장에 대해 실행 부하가 실제로 발생합니다. 운영 환경에서는 `SELECT`라도 주의해서 사용해야 하며, 데이터 변경 문장에 적용할 때는 현재 MySQL 버전에서 어떤 문장을 지원하는지 먼저 확인하는 것이 안전합니다.

Phase 6. EXPLAIN FORMAT=JSON — 비용 상세 확인

`FORMAT=JSON`은 옵티마이저가 계산한 비용을 구조적으로 보여줍니다.

```
EXPLAIN FORMAT=JSON
SELECT * FROM orders WHERE status = 'PAID';
```

```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "540.00"
    },
    "table": {
      "table_name": "orders",
      "access_type": "ref",
      "key": "idx_status",
      "rows_examined_per_scan": 1200,
      "rows_produced_per_join": 1200,
      "cost_info": {
        "read_cost": "300.00",
        "eval_cost": "240.00",
        "prefix_cost": "540.00"
      }
    }
  }
}
```

비용 구조

- `read_cost` — 데이터를 읽는 I/O 비용
- `eval_cost` — 읽은 행을 평가(필터링, 비교)하는 CPU 비용
- `prefix_cost` — 이 단계까지의 **누적** 비용
- `query_cost` — 전체 쿼리의 총 비용

이 값들을 직접 튜닝에 활용하기보다는, **두 실행 계획의 비용을 비교**하는 용도로 사용합니다. 인덱스를 추가하거나 쿼리를 변경한 뒤 `query_cost`가 줄어들었는지 확인하면 개선 효과를 정량적으로 파악할 수 있습니다.

Phase 7. 실행 계획으로 문제 찾기 — 실전 패턴

패턴 1: 인덱스를 탔는데 느린 경우

```
| type | key        | rows   | Extra       |
| ref  | idx_status | 45000  | Using where |
```

`type`이 `ref`이므로 인덱스를 사용하고 있지만, `rows`가 45,000입니다. **인덱스를 타는 것과 빠른 것은 다릅니다.** 선택도가 낮은 조건으로 인덱스를 타면 대량의 행을 읽고도 추가 필터링이 필요합니다.

개선 방향: 더 선택적인 조건을 포함한 복합 인덱스로 `rows`를 줄이거나, 커버링 인덱스를 검토합니다.

패턴 2: Using filesort + Using temporary

```
| type | key     | rows | Extra
|
| ref  | idx_uid | 500  | Using where; Using temporary;
Using filesort |
```

`GROUP BY`나 `ORDER BY`가 인덱스 순서와 맞지 않아 임시 테이블과 별도 정렬이 발생한 상태입니다.

개선 방향: `WHERE` 조건과 `ORDER BY` / `GROUP BY`를 함께 처리할 수 있는 복합 인덱스를 설계합니다. 예를 들어 `WHERE user_id = ? ORDER BY created_at`이라면 `(user_id, created_at)` 인덱스가 두 가지를 한 번에 해결합니다.

패턴 3: 조인 드리븐 테이블이 ALL

```
| id | table | type   | key        | rows  |
|  1 | o     | ref    | idx_status | 1200 |
|  1 | d     | ALL    | NULL       | 80000 |
```

드라이빙 테이블은 인덱스를 타지만, 드리븐 테이블 `d`가 풀 스캔입니다. 총 탐색량은 `1,200 × 80,000 = 9,600만`에 달할 수 있습니다.

개선 방향: 드리븐 테이블의 조인 키에 인덱스를 추가합니다. `eq_ref`나 `ref`로 바뀌면 `1,200 × 1` 수준으로 줄어듭니다.

패턴 4: 예상 rows와 실제 rows의 큰 차이

```
-- EXPLAIN:         rows=100
-- EXPLAIN ANALYZE: actual rows=25000
```

통계가 실제 데이터 분포를 반영하지 못하고 있습니다. 옵티마이저가 "100행 정도니까 이 계획이 싸다"고 판단했지만 실제로는 25,000행을 처리해야 합니다.

개선 방향: `ANALYZE TABLE`로 통계를 갱신하고 실행 계획을 다시 확인합니다. MySQL 8.0의 히스토그램(`ANALYZE TABLE ... UPDATE HISTOGRAM`)을 활용하면 데이터 분포가 편향된 컬럼의 통계를 개선할 수 있습니다.

한눈에 보는 EXPLAIN 점검 순서

실행 계획을 볼 때 아래 순서로 확인하면 문제를 빠르게 좁힐 수 있습니다.

순서	확인 항목	무엇을 보는가
1	`type`	`ALL`이나 `index`가 있는가
2	`rows`	예상 행 수가 비정상적으로 큰가
3	`key`	의도한 인덱스가 실제로 선택됐는가
4	`key_len`	복합 인덱스의 컬럼이 충분히 사용됐는가
5	`Extra`	`Using filesort`, `Using temporary`가 있는가
6	조인 순서	드라이빙 테이블의 `rows`가 적절한가
7	`EXPLAIN ANALYZE`	예상과 실제가 크게 다른 단계가 있는가

정리

1. **옵티마이저는 비용 기반으로 판단합니다** — 인덱스가 있어도 비용이 높다고 추정하면 사용하지 않습니다
2. **`type`이 가장 중요한 컬럼입니다** — `ALL`과 `index`는 개선 신호이고, `ref`라도 `rows`가 크면 주의해야 합니다
3. **`key_len`으로 복합 인덱스 활용도를 확인합니다** — 길이가 짧으면 뒤쪽 컬럼이 사용되지 않은 것입니다
4. **`Extra`에서 `Using filesort`와 `Using temporary`는 개선 후보입니다** — `WHERE`와 `ORDER BY`를 함께 처리하는 인덱스가 해결책인 경우가 많습니다
5. **조인은 드라이빙 테이블의 행 수가 성능을 결정합니다** — 첫 번째 테이블의 `rows`를 줄이는 것이 가장 효과적입니다
6. **`EXPLAIN ANALYZE`로 예상과 실제를 비교합니다** — 차이가 크면 통계 갱신이 필요합니다

PART 3

3부. 조회 성능 — 커넥션 풀 · 캐시 · 페이지네이션

Chapter 5

DB 커넥션 풀 완전 정복 — HikariCP 설정부터 풀 고갈 원인까지

#커넥션 풀

DB 커넥션 풀이 왜 필요한지, max pool size를 어떻게 봐야 하는지, 풀 고갈이 왜 발생하는지, HikariCP 핵심 옵션과 실무 점검 포인트를 정리합니다.

DB 커넥션 풀, 왜 알아야 하나요?

애플리케이션이 DB에 쿼리를 한 번 보낼 때마다 TCP 연결을 새로 만들고, 인증하고, 세션을 준비한 뒤, 작업이 끝나면 바로 끊는다고 가정해 보겠습니다. 요청이 많아질수록 어떤 일이 생길까요?

- 요청마다 연결 생성 비용이 반복됩니다
- DB가 동시에 감당해야 할 연결 수가 급격히 늘어납니다
- 응답 시간이 흔들리고, 순간적으로 장애처럼 보이는 상황이 생깁니다

이 문제를 해결하기 위해 사용하는 것이 **커넥션 풀(Connection Pool)**입니다. 커넥션 풀은 **미리 연결을 만들어 두고 재사용**해서, 연결 생성 비용을 줄이고 동시 요청을 더 안정적으로 처리하게 해줍니다.

이 글에서는 커넥션 풀이 왜 필요한지부터 시작해서, `max pool size`를 어떻게 이해해야 하는지, 풀 고갈(pool exhaustion)은 왜 생기는지, 그리고

HikariCP 기준으로 어떤 설정을 먼저 봐야 하는지까지 실무 관점에서 정리합니다.

Phase 1. 커넥션을 매번 새로 만들면 왜 비쌀까?

DB 커넥션은 단순히 "객체 하나 생성"이 아닙니다.

```
애플리케이션
  ↓
TCP 연결 생성
  ↓
DB 인증
  ↓
세션 준비
  ↓
쿼리 실행
  ↓
연결 종료
```

이 과정에는 네트워크 왕복, 인증 비용, DB 서버 리소스 할당이 모두 들어갑니다. 요청마다 이 작업을 반복하면 쿼리 자체보다 **연결 준비 비용**이 더 눈에 띄기 시작합니다.

커넥션 풀의 핵심 아이디어

커넥션 풀은 미리 여러 개의 DB 연결을 만들어 두고, 요청이 오면 그중 하나를 잠시 빌려 줍니다.

```
요청 A → 풀에서 커넥션 1 대여 → 사용 후 반납
요청 B → 풀에서 커넥션 2 대여 → 사용 후 반납
요청 C → 풀에서 커넥션 1 재사용
```

즉, 핵심은 **새로 만드는 것보다 재사용하는 것**입니다.

장점은 세 가지입니다

- **응답 시간 감소** — 이미 열린 연결을 바로 사용할 수 있습니다
- **DB 보호** — 연결 수를 제한해서 DB가 감당 가능한 범위 안에서 동작하게 합니다
- **애플리케이션 안정성 향상** — 순간 트래픽에도 연결 생성 폭증을 줄일 수 있습니다

> **참고:** 커넥션 풀은 성능을 무한히 늘려 주는 장치가 아닙니다. 더 정확히는 **DB 연결을 제한된 자원으로 관리**하게 해주는 장치입니다.

Phase 2. 커넥션 풀은 내부에서 어떻게 동작할까?

일반적인 요청 흐름은 다음과 같습니다.

```
1. 요청이 들어온다
2. 애플리케이션이 풀에 "커넥션 하나 주세요"라고 요청한다
3. 여유 커넥션이 있으면 즉시 대여한다
4. 쿼리 실행 후 커넥션을 닫는 대신 풀에 반납한다
5. 다른 요청이 그 커넥션을 다시 사용한다
```

여기서 중요한 것은 애플리케이션 코드에서 `Connection.close()`를 호출해도, 실제 물리 연결이 끊어지는 것이 아니라 **풀로 반납**된다는 점입니다.

풀이 관리하는 대표 상태

- **idle connection** — 지금은 사용 중이 아니지만 풀 안에서 대기 중인 연결
- **active connection** — 현재 누군가 빌려서 사용 중인 연결
- **pending thread/request** — 커넥션이 없어 대기 중인 요청

이 세 가지를 보면 대부분의 풀 문제를 해석할 수 있습니다.

Phase 3. `max pool size`는 무엇을 의미할까?

가장 많이 보게 되는 설정이 `maximumPoolSize`입니다. 이름은 익숙하지만, 의미를 오해하는 경우가 많습니다.

`maximumPoolSize`의 뜻

HikariCP에서 `maximumPoolSize`는 **풀이 동시에 보유할 수 있는 최대 커넥션 수**입니다.

예를 들어 `maximumPoolSize = 10`이면:

- 동시에 최대 10개의 요청만 DB 커넥션을 직접 사용할 수 있습니다
- 11번째 요청부터는 누군가 반납할 때까지 기다려야 합니다

즉, 이 값은 단순한 튜닝 숫자가 아니라 **애플리케이션이 DB에 가하는 동시성 상한**입니다.

크게 잡으면 무조건 좋을까?

아닙니다. 너무 크게 잡으면 오히려 나빠질 수 있습니다.

- DB의 `max_connections`를 빠르게 소모합니다
- 동시에 실행되는 쿼리가 많아져 DB CPU와 I/O 경합이 커집니다
- 느린 쿼리가 많을 때 문제를 숨긴 채 더 크게 폭발시킬 수 있습니다

커넥션 풀은 "클수록 좋다"가 아니라 **DB가 감당 가능한 동시 실행 수에 맞게 제한해야 하는 자원**입니다.

너무 작으면 어떤 일이 생길까?

- 요청이 풀에서 오래 대기합니다
- 애플리케이션 응답 시간이 급격히 증가합니다

- 결국 `connection timeout` 예외가 발생합니다

즉, 너무 크면 DB가 힘들고, 너무 작으면 애플리케이션이 기다립니다. 커넥션 풀 튜닝은 이 둘 사이의 균형을 맞추는 작업입니다.

Phase 4. 풀 고갈(pool exhaustion)은 왜 생길까?

실무에서 가장 자주 만나는 문제는 **커넥션 풀 고갈**입니다. 보통은 "트래픽이 많아서"라고 생각하지만, 실제 원인은 더 구체적입니다.

1. 쿼리가 느려서 커넥션을 오래 붙잡는 경우

```
요청 10개가 각각 커넥션을 하나씩 사용 중
각 요청이 3초 동안 DB 작업 수행
maximumPoolSize = 10
새 요청은 최소 3초 가까이 대기
```

풀의 크기가 10이어도, 각 요청이 커넥션을 오래 점유하면 사실상 처리량은 크게 떨어집니다.

이때 문제의 본질은 "풀이 작다"가 아니라 **커넥션 점유 시간이 길다**는 것입니다.

2. 트랜잭션이 길어서 반납이 늦는 경우

트랜잭션 격리 수준, 데이터베이스 락, MVCC 글에서 반복해서 나온 것처럼, **긴 트랜잭션은 거의 항상 비쌉니다.**

```
@Transactional
fun processOrder(orderId: Long) {
    val order = orderRepository.findById(orderId)
    val payment = externalPaymentApi.call(order)
    order.complete(payment)
}
```

이 코드에서 외부 API 호출이 트랜잭션 안에 들어가 있으면, 그 시간 동안 커넥션도 함께 점유됩니다. 외부 API가 800ms만 걸려도 고트래픽 환경에서는 풀을 빠르게 잠식할 수 있습니다.

3. 커넥션을 반납하지 못하는 경우

프레임워크를 정상적으로 사용하면 드물지만, 저수준 JDBC 코드나 비정상 흐름에서는 커넥션 누수가 생길 수 있습니다.

```
Connection con = dataSource.getConnection();
PreparedStatement ps = con.prepareStatement(sql);
ResultSet rs = ps.executeQuery();

// 예외가 발생했는데 close가 호출되지 않음
```

이런 누수가 반복되면 active connection 수는 계속 증가하고, 결국 풀 전체가 고갈됩니다.

4. 애플리케이션 스레드 수와 풀 크기 균형이 맞지 않는 경우

서버 스레드는 200개인데 풀 크기가 10이라면, 이론적으로는 동시에 190개 요청이 대기할 수 있습니다. 이것 자체가 문제는 아니지만, DB 작업 비중이 높은 서비스라면 대기 시간이 빠르게 커질 수 있습니다.

중요한 것은 **애플리케이션 스레드 수보다 커넥션 풀이 작을 수는 있지만, 그때 어떤 대기 모델을 의도하는지 알고 있어야 한다**는 점입니다.

Phase 5. HikariCP에서 먼저 봐야 할 핵심 옵션

Spring Boot에서 가장 많이 쓰는 커넥션 풀은 HikariCP입니다. 옵션은 많지만, 처음부터 전부 볼 필요는 없습니다. 우선순위가 높은 것은 몇 가지뿐입니다.

`maximumPoolSize`

풀의 최대 크기입니다.

```
spring.datasource.hikari.maximum-pool-size=20
```

가장 먼저 봐야 하는 숫자지만, 이 값만 키워서 문제를 해결하려고 하면 대부분 실패합니다. 먼저 **왜 커넥션이 오래 점유되는지**를 확인해야 합니다.

`minimumIdle`

풀 안에 최소 몇 개의 idle 커넥션을 유지할지 결정합니다.

```
spring.datasource.hikari.minimum-idle=10
```

트래픽이 갑자기 올라올 때 미리 준비된 연결이 너무 적으면 초반 응답이 흔들릴 수 있습니다. 다만 HikariCP는 일반적으로 고정 크기처럼 운용하는 경우가 많아서, 실무에서는 `minimumIdle`을 따로 크게 조정하지 않는 경우도 많습니다.

`connectionTimeout`

커넥션을 빌리기 위해 **최대 얼마나 기다릴지**를 의미합니다.

```
spring.datasource.hikari.connection-timeout=3000
```

이 시간이 지나면 보통 이런 예외를 보게 됩니다.

```
Connection is not available, request timed out after
3000ms
```

이 예외가 보인다면 "DB가 죽었다"보다 먼저 **풀이 바닥났거나, 커넥션 점유 시간이 너무 길다**고 의심하는 편이 맞습니다.

`maxLifetime`

하나의 물리 커넥션을 얼마나 오래 유지할지 정하는 값입니다.

```
spring.datasource.hikari.max-lifetime=1800000
```

DB나 프록시가 먼저 연결을 끊기 전에 애플리케이션이 더 일찍 정리하게 맞추는 용도로 사용합니다. 이 값은 DB나 인프라가 강제로 종료하는 **connection lifetime limit보다 몇 초 짧게** 잡는 것이 일반적입니다. 반대로 idle timeout 대응은 `keepaliveTime`이나 `idleTimeout`과 함께 봐야 합니다.

`idleTimeout`

오랫동안 놀고 있는 idle 커넥션을 언제 정리할지 결정합니다.

```
spring.datasource.hikari.idle-timeout=600000
```

트래픽이 들쭉날쭉한 서비스에서는 너무 공격적으로 줄이면 연결 생성과 제거가 자주 반복될 수 있습니다. HikariCP에서는 `minimumIdle < maximumPoolSize`일 때만 `idleTimeout`이 실제로 의미를 가집니다.

Phase 6. 풀 크기는 어떻게 정해야 할까?

정답 공식 하나로 끝나지는 않지만, 방향은 분명합니다.

잘못된 접근

- CPU 코어 수만 보고 결정

- 서버 스레드 수와 동일하게 맞춤
- 장애가 나면 무조건 `maximumPoolSize`만 올림

이 방식은 겉으로만 버티게 만들고, 실제 병목은 그대로 숨겨 두는 경우가 많습니다.

더 현실적인 접근

1. **DB의 최대 허용 연결 수**를 먼저 확인합니다
2. 여러 애플리케이션 인스턴스가 그 연결 수를 어떻게 나눠 쓸지 계산합니다
3. 평균 쿼리 시간과 피크 트래픽에서 active connection이 얼마나 필요한지 관찰합니다
4. 대기 시간과 DB 부하를 함께 보면서 점진적으로 조정합니다

예를 들어 DB `max_connections`가 200이고, 앱 인스턴스가 4개라면 각 인스턴스가 50씩 다 쓰는 설계는 위험합니다. 운영자 세션, 배치, 관리자 툴, 예상치 못한 연결까지 고려하면 **안전 여유분**이 필요합니다.

중요한 관점

커넥션 수는 처리량의 원인이 아니라 **결과**인 경우가 많습니다.

- 쿼리가 빠르면 적은 커넥션으로도 많은 요청을 처리할 수 있습니다
- 쿼리가 느리면 많은 커넥션을 둬도 결국 DB에서 병목이 납니다

그래서 커넥션 풀 튜닝은 단독 작업이 아니라, 인덱스 기본, 인덱스가 안 타는 이유, 실행 계획 같은 쿼리 튜닝과 함께 봐야 합니다.

Phase 7. 장애 상황에서는 무엇부터 볼까?

커넥션 풀 문제는 증상만 보면 서버가 그냥 느린 것처럼 보이기도 합니다. 이럴 때는 아래 순서로 보면 좋습니다.

1. 풀 메트릭 확인

- active connection 수가 최대치에 붙어 있는가
- idle connection이 0에 가까운가
- pending request가 계속 쌓이는가

이 세 가지가 보이면 일단 풀 고갈 가능성이 높습니다.

2. 느린 쿼리 확인

- 특정 쿼리가 오래 걸리는가
- 락 대기 때문에 쿼리가 지연되는가
- 트랜잭션이 불필요하게 길지 않은가

풀 문제의 상당수는 결국 **느린 쿼리 문제**로 돌아갑니다.

3. 애플리케이션 코드 확인

- 트랜잭션 안에서 외부 API를 호출하는가
- 파일 I/O, Redis, HTTP 호출이 DB 트랜잭션 안에 들어 있는가
- 예외 상황에서 자원이 제대로 정리되는가

4. DB 한계 확인

- DB `max_connections`에 근접했는가
- CPU, 디스크 I/O, lock wait가 급증했는가

애플리케이션 풀 크기만 키워도 되는 상황인지, 아니면 DB가 이미 한계인지 구분해야 합니다.

Phase 8. 실무에서 자주 하는 오해

오해 1. "커넥션 풀이 크면 무조건 빠르다"

아닙니다. 커넥션이 많아질수록 동시에 더 많은 쿼리가 DB로 밀려 들어갑니다. DB가 감당 가능한 범위를 넘기면 오히려 전체가 느려질 수 있습니다.

오해 2. "timeout 예외는 DB 장애다"

항상 그렇지는 않습니다. 실제로는 DB가 살아 있어도, 풀에 여유 커넥션이 없어서 애플리케이션이 기다리다가 timeout 나는 경우가 매우 많습니다.

오해 3. "풀 크기만 올리면 해결된다"

일시적으로 증상이 늦게 나타날 수는 있지만, 근본 원인이 느린 쿼리나 긴 트랜잭션이라면 결국 더 큰 규모로 다시 터집니다.

오해 4. "DB 연결은 많을수록 안전하다"

연결 수는 안전장치가 아니라 **제한 장치**에 가깝습니다. 너무 많은 연결은 보호가 아니라 과부하의 시작일 수 있습니다.

정리

1. **커넥션 풀은 DB 연결을 재사용하고 제한하는 장치입니다** — 성능 향상보다도 자원 관리 관점이 더 중요합니다
2. **`maximumPoolSize`는 애플리케이션이 DB에 거는 동시성 상한입니다** — 클수록 무조건 좋은 값이 아닙니다
3. **풀 고갈의 핵심 원인은 커넥션 점유 시간이 길다는 데 있습니다** — 느린 쿼리, 긴 트랜잭션, 외부 API 호출이 대표적입니다
4. **HikariCP에서는 `maximumPoolSize`, `connectionTimeout`, `maxLifetime`부터 봐야 합니다** — 숫자를 외우기보다 의미를 이해하는 것이 먼저입니다
5. **풀 문제는 쿼리 튜닝 문제와 연결되어 있습니다** — 인덱스, 실행 계획, 락 대기를 함께 봐야 제대로 해결됩니다

Chapter 6

N+1 쿼리 문제 완전 정복 — 왜 느려지고 어떻게 해결할까

#N+1 #쿼리 최적화

N+1 쿼리가 무엇인지, 왜 성능을 망가뜨리는지, JPA와 서비스 레이어에서 어떻게 발생하는지, 그리고 어떤 방식으로 해결해야 하는지 정리합니다.

N+1 문제, 왜 따로 알아야 하나요?

커뮤니티 모듈 성능 개선기, Discovery 모듈 성능 개선기, 채팅방 목록 요약 API 성능 개선기 같은 트러블슈팅 글을 보면 반복해서 등장하는 문제가 있습니다.

- 목록 20건인데 쿼리가 수십 번 나갑니다
- 페이지 크기를 10에서 50으로 늘렸더니 응답 시간이 갑자기 튑니다
- 캐시를 붙여도 cold start에서는 여전히 느립니다

이런 상황의 대표적인 원인이 **N+1 쿼리 문제**입니다. 이름은 단순하지만, 실무에서는 API 응답 시간, DB 부하, 커넥션 풀까지 함께 흔드는 문제입니다.

> **기준:** 이 글은 Spring Boot + JPA/Hibernate + 일반적인 서비스 레이어 조회 코드를 기준으로 설명합니다. 다만 N+1 자체는 ORM에만 있는 문제가 아니라, "목록을 읽고 루프 안에서 추가 조회를 반복하는 구조" 전반에 나타나는 문제입니다.

Phase 1. N+1 쿼리란 무엇인가?

가장 짧게 정의하면 이렇습니다.

> **기본 조회 1번 + 결과 건수 N만큼 추가 조회가 반복되는 문제**

예를 들어 주문 20건을 먼저 읽고, 각 주문의 사용자 정보를 개별 조회하면:

```
1. 주문 목록 조회 1번
2. 주문 20건 각각에 대해 사용자 조회 20번

총 21번 쿼리
```

그래서 보통 `1 + N` 구조인데, 관습적으로 **N+1 문제**라고 부릅니다.

가장 단순한 예시

```
val orders = orderRepository.findAllByStatus("PAID")    // 1번

val result = orders.map { order →
    val user = userRepository.findById(order.userId)    // N번
    OrderView(order.id, user.name)
}
```

주문이 5건이면 6번, 100건이면 101번 쿼리가 나갑니다.

핵심은 "쿼리가 여러 번 나간다" 자체보다, **조회 건수에 비례해서 쿼리 수도 선형으로 증가한다**는 점입니다.

Phase 2. N+1은 어떻게 발생할까?

N+1은 한 가지 방식으로만 생기지 않습니다. 실무에서는 크게 두 가지 패턴이 많습니다.

ORM의 지연 로딩(Lazy Loading)

JPA/Hibernate에서 가장 유명한 케이스입니다.

```
@Entity
class Order(
    @Id val id: Long,

    @ManyToOne(fetch = FetchType.LAZY)
    val user: User,
)
```

```
val orders = orderRepository.findAllByStatus("PAID")    // 1번

val views = orders.map { order →
    OrderView(
        id = order.id,
        userName = order.user.name,                      // 접근 시 추가 조회
    )
}
```

겉으로는 단순히 필드 접근처럼 보이지만, 실제로는 `order.user`를 읽는 순간 Hibernate가 프록시를 초기화하면서 추가 쿼리를 보낼 수 있습니다.

> **참고:** `EAGER`로 바꾼다고 N+1이 자동으로 사라지는 것은 아닙니다. 어떤 SQL이 나가는지는 실제 조회 쿼리와 매핑 방식에 따라 달라지므로, 결국 SQL 로그나 실행 결과로 확인해야 합니다.

```
SELECT * FROM orders WHERE status = 'PAID';      -- 1번
SELECT * FROM users WHERE id = 1;                -- N번 중 1
SELECT * FROM users WHERE id = 2;                -- N번 중 2
SELECT * FROM users WHERE id = 3;                -- N번 중 3
...
```

서비스 레이어의 루프 안 개별 조회

이건 ORM이 없어도 발생합니다.

```
val reviews = reviewStore.list(pageable)          // 1번

val result = reviews.map { review →
    val seller = sellerStore.findByItemId(review.itemId)
// N번
    val image = imageStore.findByItemId(review.itemId)
// N번
    ReviewView(review.id, seller, image)
}
```

이 구조는 JPA 프록시와 무관합니다. 그냥 **루프 안에서 조회를 반복**하고 있기 때문에 N+1입니다.

실무에서는 오히려 이 두 방식이 섞여 있는 경우가 많습니다.

- 엔티티 연관 관계 접근으로 한 번
- 외부 포트/리포지토리 호출로 또 한 번

그러면 `1 + N + N` 구조가 되어 금방 수십 번 쿼리로 커집니다.

Phase 3. 왜 이렇게 중요한가?

N+1은 단순히 "쿼리 수가 좀 많다" 수준에서 끝나지 않습니다.

응답 시간이 데이터 크기에 비례해 나빠집니다

예를 들어 목록 20건에서 건당 2개의 추가 조회가 있다면:

```
기본 목록 조회 1번
+ 판매자 조회 20번
+ 이미지 조회 20번
= 총 41번
```

페이지 크기를 50으로 올리면 바로 101번이 됩니다.

즉, 코드 한 줄이 아니라 **목록 크기 자체가 성능 문제의 레버**가 됩니다.

DB 부하와 커넥션 점유 시간이 같이 증가합니다

쿼리 1번이 빠르더라도, 그 쿼리가 수십 번 반복되면 총 시간이 커집니다.

- DB는 같은 종류의 조회를 계속 반복 처리해야 하고
- 애플리케이션은 그동안 커넥션을 더 오래 붙잡고 있고
- 동시 요청이 많아지면 커넥션 풀이 빠르게 바닥날 수 있습니다

그래서 N+1은 DB 커넥션 풀, 실행 계획 문제로도 이어집니다.

개발 환경에서는 잘 안 보입니다

로컬 DB에 데이터가 3건뿐이면:

- 1번 쿼리
- 추가 3번 쿼리

이 정도는 거의 티가 안 납니다.

하지만 운영에서는 페이지당 20건, 50건, 100건씩 조회하고 동시 요청도 많습니다. 그래서 N+1은 **초기에는 숨고, 트래픽과 데이터가 쌓일수록 터지는 문제**입니다.

캐시는 구조적 해결책이 아닙니다

캐시가 일부 조회를 가려 줄 수는 있지만, cold start나 캐시 미스가 나는 순간 N+1 구조는 다시 드러납니다. 즉, 캐시는 보강책일 수 있어도 **구조를 고치는 해결책은 아닙니다.**

Phase 4. N+1은 어떻게 찾을까?

실무에서는 보통 아래 순서로 찾습니다.

1. 페이지 크기에 따라 쿼리 수가 같이 늘어나는지 봅니다

가장 강력한 신호입니다.

```
pageSize=10  → 쿼리 11번
pageSize=20  → 쿼리 21번
pageSize=50  → 쿼리 51번
```

이 패턴이면 N+1을 강하게 의심할 수 있습니다.

2. 루프 안 조회 코드를 먼저 찾습니다

코드 리뷰에서 가장 먼저 볼 질문은 이것입니다.

> `map`, `forEach`, `associate`, `groupBy` 안에서 리포지토리나 포트를 호출하고 있지 않은가?

예:

```
items.map { item →
    sellerPort.getSeller(item)
}
```

이런 코드는 거의 항상 의심 대상입니다.

3. SQL 로그를 봅니다

SQL 로그를 켜 보면 같은 형태의 쿼리가 반복되는 경우가 많습니다.

```
select * from users where id = ?
select * from users where id = ?
select * from users where id = ?
select * from users where id = ?
```

파라미터만 바뀐 같은 조회가 연속으로 반복되면 N+1 가능성이 큽니다.

4. Lazy Loading 접근 지점을 확인합니다

JPA에서는 특히 아래 코드가 흔한 출발점입니다.

```
orders.map { it.user.name }
articles.map { it.tags.map(Tag::name) }
```

겉으로는 필드 접근처럼 보이지만, 실제로는 SQL이 뒤에서 추가로 나갈 수 있습니다.

Phase 5. 해결 방법은 무엇인가?

N+1 해결은 한 가지 기술로 끝나지 않습니다. **조회 대상의 성격**에 따라 해법이 달라집니다.

방법 1. `fetch join`으로 함께 읽는다

연관 엔티티를 한 번에 가져오는 방식입니다.

```
@Query(
    """
    select o
    from Order o
    join fetch o.user
    where o.status = :status
    """
)
fun findAllWithUserByStatus(status: OrderStatus):
List<Order>
```

이 방식은 특히 `ManyToOne`, `OneToOne` 같은 **to-one 연관 관계**에 잘 맞습니다.

장점:

- 쿼리 수를 1번으로 줄이기 쉽습니다
- 코드가 직관적입니다

주의:

- 컬렉션 `fetch join`은 결과 행이 불어나기 쉽고
- JPA에서 컬렉션 `fetch join`과 페이지네이션을 같이 쓰면 문제가 생기기 쉽습니다

방법 2. ID를 모아 `IN` 쿼리로 배치 조회한다

트러블슈팅 글들에서 가장 자주 등장한 방식입니다.

```
val itemIds = reviews.map { it.itemId }.distinct()

val sellersByItemId =
sellerRepository.findAllByItemIdIn(itemIds)
    .associateBy { it.itemId }

val imagesByItemId =
imageRepository.findAllByItemIdIn(itemIds)
    .groupBy { it.itemId }
```

그다음 메모리에서 조립합니다.

```
val result = reviews.map { review →
    ReviewView(
        id = review.id,
        seller = sellersByItemId[review.itemId],
        images = imagesByItemId[review.itemId].orEmpty(),
    )
}
```

이 방식은 아래 같은 경우에 특히 강합니다.

- 컬렉션 조회
- 외부 포트/리포지토리 반복 호출
- 서비스 레이어에서 여러 부가 정보를 조합하는 API

방법 3. DTO/Projection 조회로 N+1이 숨어들지 않게 다시 설계한다

이건 `fetch join`이나 `IN` 배치 조회처럼 연관 로딩을 직접 제어하는 기법이라기보다, **엔티티를 따라가며 추가 조회하지 않도록 조회 자체를 다시 설계하는 방식**에 가깝습니다.

목록 API라면 엔티티 그래프 전체보다 **응답에 필요한 컬럼만** 바로 읽는 편이 낫습니다.

```
@Query(
    """
    select new com.example.OrderListRow(
        o.id,
        u.name,
        o.status
    )
    from Order o
    join o.user u
    where o.status = :status
    """
)
fun findOrderListRows(status: OrderStatus):
List<OrderListRow>
```

장점:

- 필요한 값만 조회합니다
- Lazy Loading 여지를 줄입니다

- 목록/요약 API에 잘 맞습니다

즉, DTO/Projection은 "N+1을 자동으로 해결하는 기능"이라기보다, **N+1이 생기기 쉬운 엔티티 순회 구조를 아예 만들지 않는 설계**라고 보는 편이 더 정확합니다.

방법 4. `EntityGraph`를 사용한다

JPA에서 fetch 전략을 선언적으로 붙이는 방식입니다.

```
@EntityGraph(attributePaths = ["user"])
fun findAllByStatus(status: OrderStatus): List<Order>
```

장점:

- 리포지토리 메서드 수준에서 의도를 드러내기 쉽습니다

주의:

- 결국 어떤 SQL이 나가는지는 여전히 확인해야 합니다
- 복잡한 조립 API에서는 배치 조회 패턴이 더 명확할 때도 많습니다

방법 5. 배치 페치 설정으로 "완화"한다

Hibernate의 `default_batch_fetch_size`나 `@BatchSize`는 N+1을 완전히 없애기보다, **N번을 여러 묶음으로 줄이는** 전략입니다.

예를 들어 N=100인데 배치 크기가 20이면:

```
1번 + 100번
→ 1번 + 5번 정도로 완화
```

그래서 이 방법은:

- 빠른 응급처치로는 유용하지만
- 목록 API를 구조적으로 정리하는 최종 해법은 아닌 경우가 많습니다

Phase 6. 상황별로 무엇을 선택해야 할까?

정답은 하나가 아닙니다. 보통은 이렇게 판단하면 됩니다.

상황	추천 방법
`ManyToOne`, `OneToOne` 조회	`fetch join`, `EntityGraph`
컬렉션/부가 정보 조회	`IN` 배치 조회 + `Map` / `groupBy`
목록/요약 API	DTO Projection으로 조회를 다시 설계하거나, 배치 조회로 조립
기존 구조를 크게 못 바꾸는 경우	batch fetch 설정으로 완화

핵심은 이겁니다.

> **조회 결과를 한 건씩 처리하면서 추가 조회하지 말고, 필요한 데이터를 먼저 모아서 한 번에 읽는다**

Phase 7. 해결할 때 자주 하는 실수

`fetch join` 만 붙이면 끝난다고 생각하는 경우

to-one에서는 효과적이지만, 컬렉션까지 무턱대고 묶으면:

- 중복 행이 늘어나고
- 메모리 사용량이 커지고
- 페이지네이션과 충돌할 수 있습니다

캐시로 덮으려는 경우

캐시는 **해결 후 보강책**으로는 좋지만, 구조가 N+1이면:

- 캐시 미스 시 다시 터지고
- invalidation 포인트가 많아지고
- cold start 성능은 여전히 불안정합니다

트랜잭션을 길게 유지하며 Lazy Loading에 기대는 경우

이 방식은 개발 중에는 편해 보여도:

- 어디서 SQL이 나가는지 흐려지고

- API 응답 조립 과정에서 N+1이 숨어들고
- 문제를 재현하기도 어려워집니다

즉, N+1은 단순히 쿼리 수 문제가 아니라 **조회 책임이 흐려졌다는 신호**이기도 합니다.

한눈에 보는 N+1 점검 순서

실무에서는 아래 순서로 보면 대부분의 원인을 빠르게 좁힐 수 있습니다.

순서	확인 항목	무엇을 보는가
1	페이지 크기	건수가 늘 때 쿼리 수도 비례해서 느는가
2	루프 안 조회	`map`, `forEach` 안에서 조회를 호출하는가
3	Lazy 접근	`entity.user`, `entity.tags` 접근이 추가 SQL을 만드는가
4	SQL 로그	같은 형태의 조회가 파라미터만 바뀌어 반복되는가
5	해결 전략	`fetch join`, 배치 조회, DTO Projection 재설계 중 무엇이 맞는가
6	검증	개선 후 실제 쿼리 수가 줄었는가

정리

1. **N+1은 기본 조회 1번 뒤에 결과 건수 N만큼 추가 조회가 반복되는 문제입니다** — 데이터가 늘수록 쿼리 수도 선형으로 증가합니다
2. **JPA Lazy Loading만의 문제가 아닙니다** — 서비스 레이어에서 루프 안 조회를 반복해도 똑같이 발생합니다
3. **왜 중요한가?** — 응답 시간, DB 부하, 커넥션 점유 시간이 함께 커지기 때문입니다
4. **가장 먼저 찾는 방법은 페이지 크기에 따라 쿼리 수가 같이 늘어나는지 보는 것입니다**
5. **해결의 핵심은 "한 건씩 읽고 추가 조회"가 아니라 "먼저 모아서 한 번에 읽기"입니다**
6. **상황에 따라 해법이 다릅니다** — to-one은 `fetch join`, 조합형 목록 API는 배치 조회가 직접 해법인 경우가 많고, DTO Projection은 N+1이 생기지 않도록 조회 자체를 다시 설계하는 방식에 가깝습니다

Chapter 7

캐시 완전 정복 — Cache Aside부터 TTL, 무효화까지

#캐시

캐시가 왜 필요한지, 어떤 데이터를 캐시해야 하는지, Cache Aside와 쓰기 전략, TTL과 무효화를 어떻게 판단해야 하는지 실무 기준으로 정리합니다.

캐시, 왜 알아야 하나요?

같은 데이터를 계속 읽는 API는 생각보다 많습니다. 상품 상세, 홈 화면 블록, 사용자 프로필, 코드 테이블처럼 "조금 전에도 읽었던 값"을 다시 가져오는 요청이 반복됩니다.

- 조회 요청은 많은데 데이터 변경은 드뭅니다
- 같은 SQL이나 외부 API 호출이 짧은 시간 안에 반복됩니다
- DB나 원본 서비스는 이미 느린데, 트래픽까지 몰리면 더 흔들립니다

이럴 때 캐시는 응답 시간을 줄이고, 원본 저장소의 부하를 낮추고, 순간 트래픽을 흡수하는 데 큰 도움이 됩니다.

다만 캐시는 만능이 아닙니다. **캐시는 "진실의 원본"이 아니라, 자주 읽는 값을 잠시 복사해 두는 계층**입니다. 이 점을 놓치면 `TTL`을 길게만 잡거나, 무효화를 빼먹거나, 오히려 일관성을 망가뜨리기 쉽습니다.

> **기준:** 이 글은 Redis 같은 원격 캐시를 기준으로 설명합니다. 하지만 `hit`, `miss`, `TTL`, 무효화 같은 핵심 원리는 인메모리 캐시에도 거의 그대로 적용됩니다.

Phase 1. 캐시는 정확히 무엇을 해결할까?

캐시의 가장 기본적인 목적은 **비싼 읽기를 반복하지 않게 만드는 것**입니다.

예를 들어 상품 상세 API가 있다고 가정해 보겠습니다.

```
요청 1 → DB 조회 → 결과 반환
요청 2 → DB 조회 → 결과 반환
요청 3 → DB 조회 → 결과 반환
```

같은 상품을 짧은 시간 안에 여러 번 읽는다면, 원본 저장소는 같은 일을 계속 반복합니다.

캐시를 두면 흐름이 이렇게 바뀝니다.

```
요청 1 → 캐시 miss → DB 조회 → 캐시에 저장 → 결과 반환
요청 2 → 캐시 hit  → 캐시에서 바로 반환
요청 3 → 캐시 hit  → 캐시에서 바로 반환
```

이때 얻는 이점은 세 가지입니다.

- **응답 시간 감소** — 네트워크 왕복, 디스크 I/O, 복잡한 조인 비용을 반복하지 않습니다
- **원본 저장소 보호** — DB나 외부 서비스가 처리해야 할 읽기 수를 줄일 수 있습니다
- **순간 트래픽 완충** — 갑자기 요청이 몰려도 인기 데이터는 캐시가 먼저 받쳐 줍니다

`cache hit` 와 `cache miss`

실무에서 가장 많이 보게 되는 두 단어입니다.

- `cache hit` : 캐시에 값이 있어서 원본 조회 없이 바로 반환한 경우
- `cache miss` : 캐시에 값이 없어서 원본 저장소를 다시 읽은 경우

캐시는 결국 **`miss`를 얼마나 줄이느냐**의 문제입니다. 그리고 그 과정에서 **값이 얼마나 오래된 상태로 허용될 수 있느냐**를 함께 판단해야 합니다.

Phase 2. 어떤 데이터를 캐시해야 할까?

모든 읽기 데이터를 캐시할 필요는 없습니다. 캐시 대상은 보통 아래 세 조건 중 두세 개를 만족할 때 효과가 큽니다.

1. 읽기 빈도가 높다

자주 읽히는 데이터일수록 캐시 효과가 큽니다.

- 상품 상세
- 인기 게시글 목록

- 홈 화면 큐레이션 블록
- 공통 코드 테이블

반대로 거의 읽히지 않는 데이터는 캐시에 올려도 `hit`가 잘 나지 않습니다.

2. 원본 조회 비용이 비싸다

원본 저장소 조회가 비싸면 캐시 이점이 커집니다.

- 조인이 많은 SQL
- 외부 API 호출
- 집계/계산이 필요한 응답
- 여러 저장소를 조합해 만든 View 객체

3. 약간의 지연 반영이 허용된다

캐시는 본질적으로 **원본과 약간 어긋날 수 있는 구조**입니다. 그래서 변경 직후 잠시 이전 값이 보여도 직접적인 오류로 이어지지 않는 데이터와 잘 맞습니다.

반대로 이런 데이터는 신중해야 합니다.

- 결제 직후 재고 수량
- 중복 차감이 치명적인 쿠폰 수량
- 승인/권한 상태처럼 즉시성이 중요한 값

한눈에 보는 캐시 적합도

데이터 유형	캐시 적합도	이유
상품 상세, 프로필, 코드 테이블	높음	읽기가 많고 변경 빈도가 낮습니다
홈 화면 블록, 추천 결과	높음	조립 비용이 크고 약간의 지연 허용이 가능합니다
검색 결과 첫 페이지	중간	효과는 크지만 조건 조합과 무효화 범위가 넓을 수 있습니다
실시간 재고, 결제 상태	낮음	오래된 값이 직접적인 오류로 이어질 수 있습니다

핵심은 이것입니다.

> **읽기는 많고, 만들기 비싸고, 약간 늦게 반영돼도 괜찮은 값이 캐시에 잘 맞습니다**

Phase 3. 대표 전략은 무엇이고, 왜 `Cache Aside`가 자주 쓰일까?

캐시는 "어디에 저장할 것인가"보다 **언제 채우고, 언제 갱신하고, 언제 버릴 것인가**가 더 중요합니다.

가장 많이 쓰는 전략: `Cache Aside`

애플리케이션이 캐시를 직접 조회하고, 없으면 원본 저장소를 읽은 뒤 캐시에 채우는 방식입니다.

```
fun getProduct(productId: Long): ProductView {
    val cacheKey = "product:detail:$productId"
    cache.get(cacheKey)?.let { return it }

    val product =
productRepository.findViewById(productId)
    cache.put(cacheKey, product, ttl =
Duration.ofMinutes(10))
    return product
}
```

흐름은 단순합니다.

```
1. 캐시 조회
2. 있으면 바로 반환
3. 없으면 원본 조회
4. 결과를 캐시에 저장
5. 응답 반환
```

이 전략이 많이 쓰이는 이유는 명확합니다.

- 구현이 단순합니다
- 어떤 데이터에 캐시를 붙일지 애플리케이션이 직접 제어할 수 있습니다

- 처음에는 일부 API에만 선택적으로 도입하기 쉽습니다

대부분의 서비스는 `Cache Aside`부터 시작해도 충분합니다.

쓰기 시점 전략은 어떻게 다를까?

읽기 전략만 보면 반쪽입니다. 쓰기가 들어올 때 캐시를 어떻게 다룰지도 정해야 합니다.

전략	동작 방식	잘 맞는 상황
`Cache Aside`	읽을 때 miss면 채움, 쓸 때는 보통 캐시 삭제	가장 일반적인 웹 서비스 읽기 캐시
`Write Through`	애플리케이션이 캐시에 쓰고, 캐시가 원본 저장소에도 동기적으로 반영	캐시 계층이 쓰기 경로까지 함께 책임질 때
`Write Around`	DB에만 쓰고, 쓰기 시 캐시는 채우지 않음	쓰기는 많고 다시 읽힐지 불확실하며 기존 캐시는 `TTL` 또는 별도 무효화 정책으로 관리할 때

실무에서는 `Cache Aside + 변경 시 삭제(evict)` 조합이 가장 흔합니다.

> **참고:** `Write Through`는 보통 캐시가 원본 저장소 쓰기까지 함께 처리하는 구조를 가리킵니다. 애플리케이션이 DB와 캐시를 각각 직접 갱신하는 방식과는 구분해서 보는 편이 더 정확합니다.

왜 "쓰기 후 삭제"가 자주 쓰일까?

예를 들어 상품 이름을 수정한다고 가정해 보겠습니다.

```
1. 상품 이름을 DB에 반영한다
2. 트랜잭션 커밋이 끝난다
3. product:detail:{id} 캐시를 삭제한다
```

이 방식은 캐시를 직접 새 값으로 맞추는 것보다 단순합니다.

- 여러 캐시 키를 동시에 정확히 업데이트하기 어렵습니다
- 상세, 목록, 추천 결과처럼 같은 데이터를 참조하는 캐시가 여러 개일 수 있습니다
- 일단 삭제해 두면 다음 읽기에서 최신 값으로 다시 채울 수 있습니다

즉, **갱신(update)보다 삭제(evict)가 더 단순하고 안전한 경우가 많습니다**.

> **참고:** 쓰기 트랜잭션이 롤백될 수 있는 경로라면, 캐시 삭제 시점도 함께 봐야 합니다. 많은 경우 DB 반영이 확정된 뒤에 무효화하는 편이 더 안전합니다.

Phase 4. 캐시 키는 어떻게 설계해야 할까?

캐시에서 값만큼 중요한 것이 키입니다. 키 설계가 흐리면 `hit` 율이 떨어지고, 무효화 범위도 관리하기 어려워집니다.

좋은 키의 조건

- **결정적이어야 합니다** — 같은 요청이면 항상 같은 키가 나와야 합니다
- **충분히 구체적이어야 합니다** — 서로 다른 결과가 같은 키를 공유하면 안 됩니다
- **무효화 가능해야 합니다** — 나중에 어떤 범위를 지울지 예상할 수 있어야 합니다

예를 들어 이런 키는 비교적 읽기 쉽고 관리하기 좋습니다.

```
product:detail:123
user:profile:42
home:section:block:summer-sale
```

반대로 조건이 많은 목록 캐시는 더 신중해야 합니다.

```
search:q=shoes&sort=popular&page=1
```

이런 키는 얼핏 맞아 보이지만, 아래 문제가 숨어 있습니다.

- 파라미터 순서가 바뀌면 다른 키가 됩니다
- 공백, 대소문자, 기본값 포함 여부가 다르면 `hit`가 깨집니다
- 조건 조합이 너무 많으면 키 수가 폭발합니다

실무에서 자주 쓰는 보완 방법

```
data class ProductSearchCacheKey(
    val query: String,
    val sort: String,
    val page: Int,
)

fun toCacheKey(key: ProductSearchCacheKey): String =
    "product-
search:${key.query.trim().lowercase()}:${key.sort}:${key.p
age}"
```

핵심은 **요청 파라미터를 정규화한 뒤 키를 만들기**입니다.

버전 접두어도 도움이 된다

캐시 구조가 바뀌거나 직렬화 형식이 바뀌면 기존 값을 한 번에 버리고 싶을 때가 있습니다.

```
v1:product:detail:123
v2:product:detail:123
```

이렇게 버전 접두어를 두면 대규모 마이그레이션이나 직렬화 변경 시 운영이 훨씬 단순해집니다. Redis 직렬화 이슈 글도 결국 "캐시에 저장된 값의 형식"을 신경 써야 했던 사례입니다.

Phase 5. `TTL`은 어떻게 정해야 할까?

`TTL`(Time To Live)은 캐시 값이 **얼마나 오래 살아 있을지**를 정하는 만료 시간입니다.

```
cache.put("product:detail:123", product, ttl =
Duration.ofMinutes(10))
```

`TTL`을 짧게 잡으면 오래된 값이 줄어들지만 `miss`가 늘고, 길게 잡으면 `hit` 율은 좋아지지만 값이 낡을 가능성이 커집니다.

즉, `TTL`은 성능 숫자가 아니라 **일관성과 비용 사이의 타협점**입니다.

`TTL`을 정할 때 보는 기준

1. 데이터가 얼마나 자주 바뀌는가
2. 오래된 값이 얼마나 큰 문제를 만드는가
3. 원본 조회 비용이 얼마나 큰가
4. 변경 이벤트로 무효화할 수 있는가

예를 들어:

데이터	예시 TTL	생각해야 할 점
코드 테이블, 공통 설정	수십 분 ~ 수시간	자주 안 바뀌고 재조회 비용이 낮습니다
상품 상세, 프로필	수분 ~ 수십 분	변경은 가끔 있지만 조회 빈도는 높습니다
홈 화면 블록, 추천 결과	수십 초 ~ 수분	트래픽이 높고 지연 반영 허용 폭이 비교적 큽니다
재고, 가격, 권한 상태	짧게 또는 TTL 미의존	오래된 값의 위험이 커서 이벤트 무효화가 더 중요합니다

`TTL`에 정답 숫자는 없습니다. 다만 다음 원칙은 유효합니다.

- **변경이 드문 값은 길게**
- **오래된 값이 위험한 값은 짧게 또는 무효화 중심으로**
- **원본 조회가 매우 비싸면 `TTL`만 짧게 잡아서는 효과가 약합니다**

Phase 6. 무효화는 왜 중요하고, 어떻게 해야 할까?

`TTL`만 믿고 운영하면 결국 "변경 직후에 오래된 값이 계속 보인다"는 문제가 생깁니다. 이때 필요한 것이 **캐시 무효화(invalidation)** 입니다.

가장 단순한 방법: 변경 후 삭제

```
1. 사용자 프로필을 DB에 반영한다
2. 트랜잭션 커밋이 끝난다
3. user:profile:{id} 캐시를 삭제한다
```

이 방식은 구현이 간단하고, 다음 조회에서 최신 값으로 다시 채워집니다.

관련 키가 여러 개면 어떻게 할까?

여기서부터 무효화가 어려워집니다. 사용자 프로필 하나가 아래 캐시에 동시에 들어 있을 수 있습니다.

- `user:profile:42`
- `feed:card:user:42`
- `ranking:top-sellers`

그래서 무효화는 단순히 "키 하나 삭제"가 아니라 **어떤 화면과 조회 결과가 그 값을 복제하고 있는지 추적하는 문제**가 됩니다.

이런 경우에는 아래 방식이 자주 쓰입니다.

- 상세 캐시는 개별 키 삭제
- 목록/집계 캐시는 짧은 `TTL`로 타협
- 관리자 수정처럼 명확한 변경 이벤트는 메시지 기반 무효화

실제로 Discovery 캐시 글에서도 블록 수정 이벤트를 받아 선택적으로 캐시를 지우는 방식을 사용했습니다.

`TTL`과 무효화는 경쟁 관계가 아니다

둘은 보통 함께 씁니다.

- `TTL`: 언젠가는 자연스럽게 사라지게 만드는 안전망
- 무효화: 변경 직후 오래된 값을 빠르게 없애는 수단

즉, **`TTL`만으로 최신성을 보장하려고 하지 말고, 최신성이 중요할수록 무효화를 더 적극적으로 붙여야 합니다**.

Phase 7. 캐시에서 자주 망가지는 지점은 무엇일까?

캐시는 적용 자체보다 운영 중 함정이 더 많습니다.

1. 캐시를 붙였는데 `hit`가 안 나오는 경우

보통은 키 설계가 흔들린 경우가 많습니다.

- 요청마다 timestamp가 들어간다
- 정렬/필터 기본값 처리 방식이 일정하지 않다
- 사용자별로 달라야 할 응답을 공용 키로 저장한다

캐시는 "저장했다"보다 **같은 요청이 같은 키로 다시 들어오는가**가 중요합니다.

2. 캐시 미스가 한꺼번에 몰리는 경우

인기 키 `TTL`이 동시에 끝나면 여러 요청이 한꺼번에 원본 저장소로 몰릴 수 있습니다. 흔히 `cache stampede`라고 부르는 문제입니다.

```
00:00:00 인기 상품 캐시 만료
00:00:01 요청 100개가 동시에 miss
00:00:01 DB로 100개 요청이 몰림
```

이럴 때는 다음 같은 완화책을 검토합니다.

- `TTL`에 약간의 랜덤 지터를 넣기
- 아주 인기 있는 키는 선갱신(refresh)하기
- 한 요청만 재계산하고 나머지는 잠시 기다리게 하기

3. 존재하지 않는 값 때문에 원본을 계속 치는 경우

없는 상품 ID를 계속 조회하면 매번 캐시 `miss` 후 DB까지 내려갑니다. 이를 막기 위해 **없는 결과를 아주 짧게 캐시하는** **`negative caching`** 을 적용하는 경우가 있습니다.

다만 이 방식은 실제로 값이 생길 수 있는 데이터와 섞이면 주의가 필요합니다. 많은 캐시 계층은 `null` 자체를 그대로 저장하지 않거나 `miss`와 구분하기 위해 별도 sentinel 값을 사용하므로, "없음"도 데이터라는 전제를 분명히 해야 합니다.

4. 직렬화 형식이 바뀌어 기존 캐시가 깨지는 경우

캐시는 값만 저장하는 것이 아니라 **값의 형식**도 함께 운영하는 문제입니다.

- 클래스 구조 변경
- 날짜 포맷 변경
- 컬렉션 타입 변경
- 다형성 직렬화 설정 변경

이런 문제는 Redis 직렬화 이슈 글처럼 실제 장애로 이어질 수 있습니다. 그래서 캐시도 스키마처럼 다뤄야 합니다.

한눈에 보는 선택 기준

지금까지 내용을 실무 기준으로 줄이면 이렇게 정리할 수 있습니다.

질문	기본 선택
읽기가 많고 변경은 드문가	캐시 검토 가치가 큽니다
오래된 값이 치명적인가	`TTL` 보다 무효화 중심으로 봅니다
어떤 키를 지워야 할지 추적 가능한가	개별 캐시 운영이 쉬워집니다
같은 조건의 요청이 반복되는가	캐시 `hit` 가능성이 높습니다
조건 조합이 너무 많은가	목록 캐시는 범위를 좁히거나 짧은 `TTL` 로 제한합니다

정리

1. **캐시는 반복되는 비싼 읽기를 줄이는 계층입니다** — 원본 저장소를 완전히 대체하는 것이 아닙니다
2. **캐시 대상은 읽기 빈도, 조회 비용, 지연 허용 범위를 같이 봐야 합니다**
3. **대부분의 서비스는 `Cache Aside` 부터 시작해도 충분합니다** — 읽을 때 채우고, 변경 시에는 보통 삭제하는 방식이 가장 단순합니다
4. **좋은 캐시는 `TTL` 보다 키 설계와 무효화가 더 중요합니다** — `hit` 율과 운영 난이도가 여기서 갈립니다
5. **`TTL` 은 최신성을 보장하는 장치가 아니라 타협점입니다** — 최신성이 중요할수록 이벤트 기반 무효화를 함께 써야 합니다
6. **캐시는 운영 중 깨지는 지점까지 설계해야 합니다** — `cache stampede`, `negative caching`, 직렬화 변경까지 함께 봐야 실무에서 버틸 수 있습니다

Chapter 8

캐시 스탬피드 완전 정복 — 핫 키와 TTL 동시 만료가 DB를 흔들 때

#캐시

핫 키와 동시 만료 때문에 캐시 miss가 폭주할 때, TTL 지터, single flight, stale-while-revalidate, 선갱신을 어떤 기준으로 선택할지 정리합니다.

캐시 스탬피드, 왜 따로 알아야 하나요?

캐시 완전 정복 글에서 캐시는 `hit`율, `TTL`, 무효화가 중요하다고 정리했습니다. 그런데 실무에서는 캐시를 붙인 뒤에도 다음과 같은 일이 반복됩니다.

- 평소에는 빠르던 API가 특정 시각마다 갑자기 느려집니다
- Redis `miss`가 늘자마자 DB CPU와 커넥션 풀이 같이 흔들립니다
- 인기 상품, 홈 화면 블록, 랭킹 API처럼 **특정 키에만 요청이 몰립니다**

이런 문제의 대표적인 형태가 **캐시 스탬피드(cache stampede)** 입니다. 특히 요청이 많이 몰리는 **핫 키(hot key)** 와 만나면, 캐시는 완충재가 아니라 병목 증폭기가 될 수 있습니다.

> **기준:** 이 글은 Redis 같은 원격 캐시와 일반적인 `Cache Aside` 구조를 기준으로 설명합니다. 라이브러리와 인프라에 따라 락 구현이나 stale 처리 방식은 달라질 수 있습니다.

이 글의 핵심은 세 가지 판단입니다.

- 언제 `TTL` 지터만으로 충분한가
- 언제 `single flight`가 필요하고, 언제 `stale-while-revalidate`가 더 나은가
- 선갱신과 `negative caching`은 어떤 경우에만 써야 하는가

먼저 적용 순서부터 보면

운영 중인 서비스에서 캐시 스탬피드가 의심된다면, 보통은 아래 순서가 가장 안전합니다.

1. **키별 `miss` 율과 재생성 시간을 먼저 본다** - 어떤 키가 진짜 핫 키인지 식별합니다
2. **전체 만료 파동부터 줄인다** - 대부분은 `TTL` 지터만으로도 큰 스파이크가 완화됩니다
3. **소수 핫 키만 별도 보호한다** - `single flight`, per-key lock, `stale-while-revalidate`를 검토합니다
4. **항상 뜨거운 공용 키만 선갱신한다** - 모든 키에 선갱신을 거는 것은 대개 과합니다

핵심은 처음부터 복잡한 락 구조를 넣지 않는 것입니다. **만료 분산 -> 핫 키 식별 -> 선택적 보호** 순서가 보통 더 잘 맞습니다.

Phase 1. 캐시 스탬피드는 정확히 무엇인가?

> **여러 요청이 같은 시점에 같은 캐시 miss를 만나 원본 저장소로 한꺼번에 몰리는 현상**

예를 들어 홈 화면 인기 상품 블록 캐시가 10분마다 만료된다고 가정해 보겠습니다.

```
09:59:59  cache hit
10:00:00  TTL 만료
10:00:01  요청 200개가 동시에 cache miss
10:00:01  DB 또는 원본 API로 200개 요청이 몰림
10:00:02  응답 시간 급증, 에러율 증가
```

핵심은 `miss` 자체가 아니라 **같은 값을 여러 요청이 동시에 다시 만들려 한다**는 점입니다.

일반적인 캐시 miss와 뭐가 다를까?

- 일반 miss: 한두 요청이 캐시에 없어서 원본 조회 후 다시 채움
- 캐시 스탬피드: 같은 키를 향한 다수 요청이 동시에 miss를 만나 원본 조회를 중복 수행

즉, 스탬피드는 **"miss 자체"보다 "동시성"** 이 핵심입니다.

핫 키는 왜 같이 언급될까?

핫 키는 **하나의 캐시 키에 요청이 유난히 많이 몰리는 상태**를 말합니다.

- `home:ranking`
- `product:detail:123`
- `event:landing:summer`
- `user:profile:1` 같은 매우 유명한 계정

이런 키는 평소에는 캐시 효과가 크지만, 만료되는 순간 가장 위험합니다. 같은 키를 기다리던 요청이 많기 때문입니다.

> **정리:** 핫 키는 "요청 집중" 문제이고, 캐시 스탬피드는 "동시 miss 폭주" 문제입니다. 둘은 다른 개념이지만 실무에서는 자주 함께 나타납니다.

Phase 2. 단순한 `Cache Aside` 구현은 왜 쉽게 터질까?

가장 흔한 `Cache Aside` 코드는 보통 이런 형태입니다.

```
fun getPopularProducts(): PopularProductsView {
    val cacheKey = "home:popular-products"
    cache.get(cacheKey)?.let { return it }

    val result = productRepository.findPopularProducts()
    cache.put(cacheKey, result, ttl =
Duration.ofMinutes(10))
    return result
}
```

평소에는 문제없어 보여도 `TTL`이 끝나는 순간에는 아래처럼 동작할 수 있습니다.

```
요청 A → miss → DB 조회
요청 B → miss → DB 조회
요청 C → miss → DB 조회
...
요청 Z → miss → DB 조회
```

즉, **캐시 하나를 다시 채우기 위해 같은 원본 조회가 중복 실행**됩니다.

왜 이렇게 위험할까?

- 비싼 SQL이 동시에 여러 번 실행됩니다
- 외부 API라면 rate limit에 걸릴 수 있습니다
- DB 커넥션이 짧은 시간 안에 고갈될 수 있습니다
- 느려진 응답이 다시 재시도를 불러 더 큰 폭주로 이어질 수 있습니다

특히 원본 조회 시간이 길수록 더 위험합니다.

```
재생성 시간 50ms  → 잠깐의 스파이크로 끝날 수 있음
재생성 시간 2s    → 2초 동안 동시 요청이 계속 누적됨
```

결국 캐시 스탬피드는 **재생성 시간이 긴 값을 동시에 다시 만드는 구조**에서 커집니다.

Phase 3. 왜 `TTL`이 같으면 더 자주 터질까?

캐시 스탬피드는 핫 키 하나 때문에만 생기지 않습니다. **여러 키가 비슷한 시점에 함께 만료되는 구조**도 흔한 원인입니다.

예를 들어 배치나 워밍업 과정에서 다음처럼 저장했다고 가정해 보겠습니다.

```
12:00:00 상품 상세 1번 저장, TTL 10분
12:00:00 상품 상세 2번 저장, TTL 10분
12:00:00 상품 상세 3번 저장, TTL 10분
...
12:10:00 인기 키 다수 동시 만료
```

이 구조에서는 사용자가 많은 시간대마다 만료 파동이 반복될 수 있습니다.

실무에서 자주 보이는 원인

- 모든 캐시에 **고정 TTL**만 사용
- 배치로 대량 적재하면서 **만료 시각이 정렬됨**

- 애플리케이션 재시작 직후 워밍업한 키들이 **같은 주기로 만료**
- 인기 이벤트 페이지처럼 **짧은 시간 동안 특정 키 트래픽이 집중**

즉, 만료는 개별 키 문제가 아니라 **전체 키 스케줄링 문제**가 되기도 합니다.

Phase 4. 가장 먼저 적용할 완화책: `TTL` 지터(jitter)

가장 단순한 첫 대응은 **만료 시간을 조금씩 흔들어 놓는 것**입니다.

예를 들어 10분 TTL을 고정하지 않고, 9분 30초에서 10분 30초 사이로 랜덤하게 분산합니다.

```
fun randomTtl(baseSeconds: Long): Duration {
    val jitter = ThreadLocalRandom.current().nextLong(-30,
31)
    return Duration.ofSeconds(baseSeconds + jitter)
}

cache.put(cacheKey, value, ttl = randomTtl(600))
```

핵심은 **동시에 만료될 키를 시간축에서 흩뜨리는 것**입니다.

장점

- 구현이 단순합니다
- 거의 모든 캐시 계층에서 적용 가능합니다
- 대량 키 동시 만료 문제를 꽤 완화합니다

한계

- **같은 키에 대한 동시 miss** 자체를 막지는 못합니다
- 재생성 시간이 긴 핫 키에는 이것만으로 부족할 수 있습니다

지터는 좋은 기본값이지만, **핫 키 하나의 동시 miss**까지 막아 주지는 못합니다.

Phase 5. 핵심 완화책: 한 요청만 재생성하게 만들기

> **같은 키를 다시 채우는 작업은 동시에 여러 번 하지 말고, 한 번만 하자**

이 접근을 흔히 `single flight`, `request coalescing`, `per-key lock` 같은 이름으로 부릅니다.

동작 흐름

```
1. 요청 A가 miss를 만난다
2. 요청 A만 재생성 권한을 얻는다
3. 요청 B, C, D는 잠깐 기다리거나 기존 stale 값을 받는다
4. 요청 A가 새 값을 캐시에 저장한다
5. 다른 요청은 새 값을 사용한다
```

예시: per-key lock 기반 흐름

```
fun getPopularProducts(): PopularProductsView {
    val cacheKey = "home:popular-products"
    cache.get(cacheKey)?.let { return it }

    val lockKey = "lock:$cacheKey"
    val token = UUID.randomUUID().toString()

    if (cache.setIfAbsent(lockKey, token, ttl =
Duration.ofSeconds(5))) {
        try {
            cache.get(cacheKey)?.let { return it }

            val result =
productRepository.findPopularProducts()
            cache.put(cacheKey, result, ttl =
Duration.ofMinutes(10))
            return result
        } finally {
            cache.deleteIfValueMatches(lockKey, token)
        }
    }

    Thread.sleep(50)
    return cache.get(cacheKey) ?:
productRepository.findPopularProducts()
}
```

위 코드는 개념 설명용입니다. 실제 운영에서는 아래를 함께 봐야 합니다.

- 락 TTL이 너무 짧으면 재생성 중에 락이 풀릴 수 있습니다
- 락 TTL이 너무 길면 장애 시 대기가 길어질 수 있습니다
- 락 해제는 **내가 잡은 락인지 확인**하고 풀어야 합니다
- 멀티 인스턴스 환경에서는 로컬 락이 아니라 **공유 락 저장소**가 필요할 수 있습니다
- `Thread.sleep`처럼 스레드를 묶는 방식은 설명용일 뿐이고, 실제로는 짧은 backoff, 제한된 재시도, 비동기 대기 중 하나를 더 많이 씁니다

이 방식이 특히 잘 맞는 경우

- 같은 키 재생성 비용이 큽니다
- 핫 키 요청량이 높습니다
- 약간 기다리더라도 중복 원본 조회를 막는 편이 낫습니다

이 방식이 과할 수 있는 경우

- 대부분의 키가 자주 안 읽힙니다
- 재생성 비용이 매우 작습니다
- 락 자체의 복잡도와 운영 비용이 더 큽니다

핵심은 **모든 키를 락으로 감쌀지보다, 정말 뜨거운 키 몇 개를 정확히 식별했는가**입니다.

기다리게 할까, 오래된 값을 줄까?

핫 키를 보호할 때는 결국 두 가지 중 하나를 고르게 됩니다.

질문	더 잘 맞는 선택
오래된 값을 보여주면 안 되는가	`single flight` 나 짧은 대기 기반 보호
몇 초 정도 stale 값을 보여줘도 되는가	`stale-while-revalidate`
재생성 비용이 크지만 요청은 기다릴 수 있는가	`single flight`
응답 지연보다 빠른 반환이 더 중요한가	`stale-while-revalidate`

두 전략의 차이는 구현 취향보다 **최신성과 지연 시간 중 무엇을 더 우선하는가**에 가깝습니다.

Phase 6. 기다리게 하지 않고 넘기는 방법: `stale-while-revalidate`

아주 잠깐 오래된 값을 보여줘도 되는 데이터라면, 요청을 세우기보다 **예전 값을 먼저 주고 뒤에서 새로 고치는 방식**이 낫습니다. 이것이 `stale-while-revalidate` 패턴입니다.

핵심 아이디어

- **fresh TTL** 안에서는 정상 캐시처럼 바로 반환
- **stale 허용 구간**에서는 이전 값을 반환하면서 비동기로 갱신
- **hard TTL** 이후에는 더 이상 stale 값을 쓰지 않고 재생성

```
freshUntil = 10:00
staleUntil = 10:05

09:59 → 최신 값 반환
10:02 -> 오래된 값 반환 + 백그라운드 갱신
10:06 -> stale도 불가, 재생성 필요
```

언제 잘 맞을까?

- 홈 화면 블록
- 인기 검색어
- 추천 결과
- 랭킹, 배너, 콘텐츠 큐레이션

즉, **몇 초~몇 분 정도의 지연 반영이 치명적이지 않은 읽기**에 잘 맞습니다.

장점

- 사용자 요청을 거의 막지 않습니다
- 핫 키 재생성 타이밍의 응답 시간 급증을 완화합니다
- 백엔드 스파이크를 줄이기 좋습니다

주의점

- 오래된 값을 허용할 수 없는 데이터에는 맞지 않습니다
- stale 허용 기간이 너무 길면 사실상 무효화가 느슨해집니다

- 비동기 갱신 실패 시 재시도 정책도 함께 봐야 합니다

이 패턴은 **정확히 최신일 필요는 없지만, 빠르게 보여주는 것이 더 중요한 값**에 잘 맞습니다.

Phase 7. 정말 뜨거운 키는 미리 갱신할 수 있다

매우 자주 읽히는 키라면, 만료 뒤에 다시 채우기를 기다리기보다 **만료 직전에 선갱신(refresh ahead)** 하는 편이 낫습니다.

예를 들어 이런 식입니다.

```
랭킹 캐시 TTL 10분
만료 1분 전부터 비동기 갱신 예약
실제 만료 전에 새 값으로 교체
```

잘 맞는 대상

- 메인 페이지 상단 블록
- 짧은 주기로 갱신되는 랭킹
- 트래픽이 항상 높은 공용 키

장점

- 사용자 요청 경로에서 재생성 비용을 뺄 수 있습니다
- 만료 순간의 스파이크를 줄이기 좋습니다

단점

- 실제로 안 읽히는 키까지 갱신하면 낭비가 큽니다
- 스케줄러나 워커 운영이 필요할 수 있습니다
- 갱신 실패 감지와 fallback 흐름을 따로 설계해야 합니다

선갱신은 **모든 키를 위한 기본 전략**이 아니라, **정말 읽기 집중도가 높은 소수 키에만 쓰는 최적화**에 가깝습니다.

Phase 8. 존재하지 않는 값도 핫 키가 될 수 있다

스탬피드는 "있는 값이 만료되는 경우"에만 생기지 않습니다. **원래부터 없는 값**을 반복 조회하는 경우도 위험합니다.

예를 들어:

- 존재하지 않는 상품 ID를 계속 조회
- 삭제된 게시글 URL이 외부에 널리 퍼짐
- 잘못된 사용자 ID를 봇이 계속 호출

이 경우 요청은 매번 이런 경로를 탑니다.

```
cache miss → DB 조회 -> 없음 -> 응답 반환
cache miss → DB 조회 -> 없음 -> 응답 반환
cache miss → DB 조회 -> 없음 -> 응답 반환
```

이때는 **negative caching** 이 도움이 됩니다. "없음"도 아주 짧게 캐시하는 방식입니다.

```
val value = repository.findProduct(productId)
if (value == null) {
    cache.put(cacheKey, NotFoundSentinel, ttl =
Duration.ofSeconds(30))
    return null
}
```

주의점

- 나중에 값이 생길 수 있는 도메인이라면 TTL을 짧게 둬야 합니다
- `null` 과 "없음 sentinel"을 구분해야 합니다
- 잘못된 키 폭주 자체를 막으려면 rate limiting도 함께 고려해야 합니다

즉, 핫 키는 인기 데이터만의 문제가 아니라, **반복되는 잘못된 요청**에서도 생길 수 있습니다.

Phase 9. 장애 상황에서는 무엇을 먼저 볼까?

캐시 스탬피드는 로그 한 줄보다 **함께 뛰는 지표 묶음**으로 보는 편이 정확합니다.

자주 같이 뛰는 지표

- 캐시 `miss` 비율 급증

- 특정 키의 요청량 급증
- 원본 DB QPS 또는 외부 API 호출 수 급증
- 애플리케이션 `p95`, `p99` 응답 시간 상승
- 커넥션 풀 active/pending 수 증가
- 타임아웃, 재시도, 서킷 브레이커 open 증가

패턴으로 보면 더 명확하다

```
10:00:00 cache expired 증가
10:00:01 cache miss 급증
10:00:01 DB QPS 급증
10:00:02 Hikari pending 증가
10:00:03 API p99 급등
```

이 흐름이 보이면 "DB가 갑자기 느려졌다"보다 먼저 **캐시 만료 타이밍과 핫 키 존재 여부**를 의심하는 편이 맞습니다. 특히 DB 커넥션 풀 글에서 본 active/pending 증가가 같은 시각에 보인다면, 스탬피드가 커넥션 풀 고갈로 번지고 있을 가능성이 큽니다.

운영에서 특히 유용한 질문

1. 어떤 키가 가장 많이 `miss` 났는가
2. 그 키의 재생성 시간은 얼마나 걸리는가
3. 같은 시각에 함께 만료된 키가 많았는가
4. stale 허용이 가능한 데이터인가
5. 락이나 `single flight` 없이 중복 재생성이 일어나고 있는가

Phase 10. 어떤 전략을 언제 골라야 할까?

정답은 하나가 아니라, 데이터 성격에 따라 전략을 조합하는 쪽에 가깝습니다.

상황	기본 선택
여러 키가 비슷한 시각에 만료된다	`TTL` 지터부터 적용
소수 핫 키의 재생성 비용이 크다	`single flight` 또는 per-key lock 검토
약간 오래된 값 허용 가능	`stale-while-revalidate` 가 유리
매우 자주 읽히는 공용 키다	선갱신(refresh ahead) 검토
없는 값 조회가 반복된다	`negative caching` 검토

결론만 다시 압축하면 아래처럼 볼 수 있습니다.

구분	먼저 볼 선택
기본값	`Cache Aside + TTL` 지터
핫 키 보호	`single flight` 또는 `stale-while-revalidate`
예외 상황	선갱신, `negative caching`

실무에서 자주 쓰는 조합

- **기본값:** `Cache Aside + TTL 지터`

- **핫 키 추가 대응:** `TTL 지터 + single flight`
- **사용자 체감 응답 우선:** `TTL 지터 + stale-while-revalidate`
- **초고빈도 공용 키:** `TTL 지터 + 선갱신 + 필요시 single flight`

> **모든 키를 같은 방식으로 다루지 말고, 핫 키와 일반 키를 구분해야 합니다**

캐시 전략은 보통 "기능별"보다 **키의 트래픽 분포와 재생성 비용**에 따라 갈립니다.

한눈에 보는 안티패턴

아래는 캐시 스탬피드를 더 쉽게 만드는 대표 패턴입니다.

- 모든 키에 동일한 고정 `TTL` 만 적용한다
- 핫 키를 식별하지 않고 전부 같은 정책으로 운영한다
- 오래된 값을 허용할 수 없는데도 `TTL` 만 믿고 간다
- 반대로 오래된 값을 허용해도 되는데 무조건 동기 재생성만 한다
- 락을 도입했지만 락 만료, 중복 해제, 장애 시 복구 흐름을 설계하지 않는다

정리

1. **실무 기본값은 `Cache Aside + TTL` 지터입니다** - 대부분의 만료 파동은 여기서 먼저 줄어듭니다

2. **정말 뜨거운 키만 별도 보호합니다** - 최신성이 중요하면 `single flight`, 응답 속도가 더 중요하면 `stale-while-revalidate`를 고릅니다
3. **선갱신과 `negative caching`은 선택 카드입니다** - 항상 뜨거운 공용 키나 반복되는 잘못된 요청처럼 분명한 대상이 있을 때만 씁니다

Chapter 9

페이지네이션 완전 정복 — OFFSET/LIMIT이 느려지는 이유와 커서 기반 조회 설계

#쿼리 최적화 #인덱스

OFFSET/LIMIT 페이지네이션이 왜 뒤로 갈수록 느려지는지, 커서 기반 조회는 어떤 조건에서 유리한지, 정렬과 인덱스를 어떻게 설계해야 하는지 실무 기준으로 정리합니다.

페이지네이션, 왜 따로 알아야 하나요?

목록 API를 만들다 보면 페이지네이션은 거의 항상 들어갑니다. 게시글 목록, 상품 목록, 채팅방 목록, 관리자 화면까지 대부분의 조회 API가 페이지 단위 응답을 사용합니다.

그런데 실무에서는 이런 문제가 자주 생깁니다.

- 초반 페이지는 빠른데 뒤로 갈수록 응답이 느려집니다
- `OFFSET` 기반 조회를 쓰는데 데이터가 중간에 끼거나 빠진 것처럼 보입니다
- 무한 스크롤을 붙이려는데 다음 페이지 기준을 어떻게 잡아야 할지 애매합니다
- 정렬 조건은 있는데 어떤 인덱스를 만들어야 할지 감이 안 잡힙니다

- 총 개수까지 같이 내려주려니 `COUNT` 쿼리 비용도 부담됩니다

페이지네이션은 단순히 `LIMIT 20 OFFSET 40`을 붙이는 문제가 아닙니다. **정렬 기준, 인덱스 구조, 데이터 변경 시 일관성, UX 방식**이 함께 얽힌 설계 문제입니다.

> **기준:** 이 글은 MySQL 8.x와 일반적인 API 서버 조회 패턴을 기준으로 설명합니다. 다만 페이지네이션의 핵심 원리는 다른 DBMS에도 거의 동일하게 적용됩니다.

먼저 선택 기준부터 보면

실무에서는 보통 아래처럼 판단하면 크게 틀리지 않습니다.

상황	더 잘 맞는 방식	이유
관리자 화면, 페이지 번호 이동이 중요함	`OFFSET/LIMIT`	구현이 단순하고 `page=37` 같은 직접 이동이 쉽습니다
무한 스크롤, 피드, 대용량 목록	커서 기반	깊은 페이지로 갈수록 더 안정적이고 빠릅니다
총 개수와 마지막 페이지 번호가 꼭 필요함	`OFFSET/LIMIT` 또는 별도 `COUNT`	커서 방식은 "몇 페이지째" 개념과 잘 안 맞습니다
데이터가 자주 추가되고 앞쪽 정렬이 흔들림	커서 기반	앞 페이지에 새 데이터가 끼어들 때 중복/누락을 줄이기 쉽습니다

핵심은 `OFFSET`이 틀렸느냐가 아닙니다. **UI와 데이터 규모에 맞는 방식을 고르는 것**이 중요합니다.

Phase 1. 페이지네이션은 정확히 무엇을 하는가?

페이지네이션은 결국 **정렬된 결과 집합에서 일부 구간만 잘라서 반환하는 것**입니다.

그래서 페이지네이션에서 가장 먼저 정해야 할 것은 `LIMIT`이 아니라 **정렬 기준(`ORDER BY`)** 입니다.

예를 들어 게시글 최신순 목록이라면 보통 이렇게 생각합니다.

```
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
ORDER BY created_at DESC
LIMIT 20;
```

겉으로는 충분해 보이지만, 여기에는 한 가지 빈틈이 있습니다.

- 같은 초에 생성된 게시글이 여러 개면 순서가 애매할 수 있습니다
- 요청마다 같은 `created_at` 묶음의 내부 순서가 바뀌면 페이지 경계가 흔들릴 수 있습니다

그래서 실무에서는 보통 **동률을 깨 줄 `tie-breaker`** 를 함께 둡니다.

```
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

즉, 페이지네이션의 출발점은 이것입니다.

> **항상 재현 가능한 정렬 순서를 먼저 만든다**

이 원칙이 깨지면 `OFFSET`이든 커서든 안정적인 페이지네이션이 어렵습니다.

Phase 2. `OFFSET/LIMIT` 방식은 어떻게 동작할까?

가장 익숙한 방식은 `OFFSET/LIMIT`입니다.

```
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
ORDER BY created_at DESC, id DESC
LIMIT 20 OFFSET 0;
```

다음 페이지는 이렇게 됩니다.

```sql
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
ORDER BY created_at DESC, id DESC
LIMIT 20 OFFSET 20;
```

세 번째 페이지는 `OFFSET 40`, 네 번째는 `OFFSET 60`입니다.

장점

- 구현이 단순합니다
- `page=1`, `page=2`, `page=37`처럼 페이지 번호 기반 UI와 잘 맞습니다
- 정확한 총 개수와 함께 내려주기 쉽습니다
- 데이터 규모가 작거나 앞쪽 몇 페이지만 보는 화면에서는 충분히 실용적입니다

잘 맞는 화면

- 관리자 목록
- 백오피스 검색 결과
- 페이지 번호 버튼이 중요한 게시판
- 사용자가 임의 페이지로 점프해야 하는 화면

즉, `OFFSET/LIMIT`은 지금도 충분히 유효한 도구입니다. 문제는 **깊은 페이지와 큰 테이블**로 갈수록 비용이 커진다는 점입니다.

Phase 3. 왜 `OFFSET`은 뒤로 갈수록 느려질까?

예를 들어 20개씩 보여 주는 목록에서 5001번째 페이지를 읽는다고 가정해 보겠습니다.

```
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
ORDER BY created_at DESC, id DESC
LIMIT 20 OFFSET 100000;
```

이 쿼리는 결과 20건만 반환하지만, DB 입장에서는 **앞의 100000건을 먼저 지나가야** 합니다.

핵심은 이것입니다.

- `LIMIT 20`은 "20건만 반환"하라는 뜻입니다
- `OFFSET 100000`은 "그 전에 있는 100000건은 건너뛰라"는 뜻입니다
- 즉, DB는 보통 **100020건 위치까지 도달한 뒤에야** 원하는 20건을 줄 수 있습니다

인덱스가 있더라도 사정이 완전히 달라지지는 않습니다.

- 정렬에 맞는 인덱스가 있으면 **정렬 비용은 줄어들 수 있습니다**
- 하지만 깊은 `OFFSET`에서는 여전히 **앞쪽 인덱스 엔트리를 많이 스캔**해야 합니다

- 선택 컬럼이나 추가 조건에 따라 테이블 접근 비용까지 붙으면 더 비싸질 수 있습니다

즉, `OFFSET`의 본질적인 비용은 **앞쪽 구간을 버리기 위해 읽는 작업**에 있습니다.

페이지 크기가 커지면 더 민감해진다

페이지 크기를 20에서 100으로 늘리면 어떻게 될까요?

- 한 번에 읽는 행 수가 늘어납니다
- 뒤쪽 페이지의 스캔 비용도 더 커집니다
- 페이지 크기 증가가 N+1 쿼리 문제와 겹치면 응답 시간은 더 급격히 흔들릴 수 있습니다

그래서 "페이지네이션이 느리다"는 말은 종종 페이지네이션 자체보다 **큰 페이지 크기 + 깊은 `OFFSET` + 추가 조회 구조**가 합쳐진 결과이기도 합니다.

Phase 4. 정렬과 인덱스가 페이지네이션 성능을 좌우한다

페이지네이션 쿼리에서 인덱스가 중요한 이유는 단순합니다.

> **정렬과 필터를 인덱스가 뒷받침하지 못하면, 페이지네이션은 앞 페이지조차 불필요하게 비싸질 수 있습니다**

예를 들어 이런 쿼리가 있다고 가정해 보겠습니다.

```
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

이 경우 실무에서는 보통 아래와 같은 복합 인덱스를 **유력한 후보**로 먼저 떠올립니다.

```
(status, created_at DESC, id DESC)
```

이 쿼리 패턴에서 이 인덱스가 자주 잘 맞는 이유는 다음과 같습니다.

- `status = 'PUBLISHED'`로 먼저 범위를 좁히고
- 그 안에서 `created_at DESC, id DESC` 순서대로 읽을 수 있기 때문입니다

자주 쓰는 출발점

1. **동등 조건 컬럼**을 먼저 검토합니다
2. 그다음 **정렬 컬럼**을 붙이는 후보를 봅니다
3. 동률을 깨는 **`tie-breaker` 컬럼**도 인덱스에 포함할지 봅니다

예:

```
WHERE author_id = ?
  AND status = 'PUBLISHED'
ORDER BY created_at DESC, id DESC
```

이런 쿼리라면 보통 아래 같은 인덱스를 대표 후보로 검토합니다.

```
(author_id, status, created_at DESC, id DESC)
```

다만 이 순서를 절대 규칙처럼 받아들이면 안 됩니다. 실제 인덱스 선택은 데이터 분포, 선택도, 추가 조건, 조회 컬럼, MySQL 옵티마이저 판단에 따라 달라질 수 있으므로 최종적으로는 `EXPLAIN`으로 확인해야 합니다.

자주 놓치는 함정

- `ORDER BY DATE(created_at)`처럼 정렬 컬럼에 함수를 씌움
- 정렬과 맞지 않는 인덱스를 기대함
- `tie-breaker` 없이 `created_at` 하나만 정렬에 사용함
- `EXPLAIN` 없이 "인덱스가 있으니 괜찮겠지"라고 생각함

이 부분은 인덱스 기본 글, 인덱스가 안 타는 이유, 실행 계획 읽는 법과 함께 보면 더 잘 연결됩니다.

Phase 5. 커서 기반 페이지네이션은 무엇이 다를까?

커서 기반 페이지네이션은 흔히 **`keyset pagination`** 또는 **`seek method`** 라고도 부릅니다.

핵심 아이디어는 단순합니다.

> **"몇 건을 건너뛸지"가 아니라 "마지막으로 본 위치가 어디인지"를 기준으로 다음 페이지를 읽는다**

예를 들어 첫 페이지는 이렇게 읽습니다.

```
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

이 페이지의 마지막 행이 다음 값이었다고 가정해 보겠습니다.

```
created_at = 2026-04-10 10:00:00
id         = 105
```

그다음 페이지는 `OFFSET 20`이 아니라, 이 마지막 값을 기준으로 읽습니다.

```sql
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
  AND (
    created_at < '2026-04-10 10:00:00'
    OR (created_at = '2026-04-10 10:00:00' AND id < 105)
  )
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

설명을 위해 조건을 풀어 썼지만, 의미는 명확합니다.

- 이전 페이지의 마지막 행보다 **뒤에 있는 데이터만** 읽습니다
- 앞쪽 20건, 40건, 100000건을 버리기 위해 스캔하지 않습니다

왜 깊은 페이지에서 유리할까?

커서 방식은 보통 "이 위치 다음부터" 읽는 형태이기 때문에, 깊은 `OFFSET` 처럼 앞 구간을 반복해서 버리는 비용이 적습니다.

그래서 특히 아래 상황에서 잘 맞습니다.

- 무한 스크롤
- 최신순 피드
- 채팅방 목록
- 오래된 페이지까지 계속 내려가는 대용량 목록

즉, 커서 방식의 강점은 **깊은 페이지 접근과 안정적인 이어 읽기**에 있습니다.

Phase 6. 커서 컬럼은 어떻게 골라야 할까?

좋은 커서 컬럼은 아무 컬럼이나 될 수 없습니다. 보통 아래 조건이 중요합니다.

- **정렬 기준과 같아야 합니다** - 커서는 결국 정렬 경계를 표현해야 합니다
- **거의 변하지 않아야 합니다** - 중간에 값이 바뀌면 레코드 위치가 움직입니다
- **순서를 안정적으로 구분할 수 있어야 합니다** - 동률이 많다면 보조 키가 필요합니다
- **인덱스로 지원돼야 합니다** - 커서 조건도 결국 검색 조건이기 때문입니다

자주 쓰는 조합

정렬 목적	자주 쓰는 커서
최신순	`created_at + id`
오래된 순	`created_at + id`
ID 순 탐색	`id`
점수 + 최신순	`score + created_at + id`

피해야 할 예시

- 자주 수정되는 `updated_at` 하나만 커서로 사용

- 중복이 많은 값 하나만 사용
- 제목, 이름처럼 표시용 문자열 사용
- `NULL`이 섞여 정렬 의미가 불명확한 컬럼 사용

중요한 실무 포인트

커서는 **정렬과 필터에 종속**됩니다.

예를 들어:

- `status = 'PUBLISHED'` 최신순 목록에서 만든 커서
- `status = 'DRAFT'` 목록에 그대로 재사용

이렇게 하면 안 됩니다. 커서는 "이 조건에서의 다음 위치"를 의미하므로, **정렬이나 필터가 바뀌면 기존 커서는 무효**입니다.

Phase 7. 왜 `created_at` 하나만으로는 부족할까?

이 부분이 커서 기반 페이지네이션에서 가장 많이 빠지는 함정입니다.

예를 들어 첫 페이지 마지막 행이 다음과 같다고 해 보겠습니다.

```
(created_at = 2026-04-10 10:00:00, id = 105)
```

그런데 아직 같은 시각의 다른 행이 더 남아 있습니다.

```
(2026-04-10 10:00:00, 104)
(2026-04-10 10:00:00, 103)
```

이때 다음 페이지 조건을 이렇게 쓰면 문제가 생깁니다.

```
WHERE created_at < '2026-04-10 10:00:00'
```

그러면 `104`, `103`은 **같은 시각이기 때문에 통째로 빠집니다.**

반대로 이렇게 쓰면:

```
WHERE created_at ⩽ '2026-04-10 10:00:00'
```

이번에는 `105`가 **다시 포함되어 중복**될 수 있습니다.

그래서 `tie-breaker`가 필요합니다.

```
WHERE created_at < :cursorCreatedAt
   OR (created_at = :cursorCreatedAt AND id < :cursorId)
```

즉, 커서 기반 페이지네이션은 사실상 아래 두 가지를 함께 요구합니다.

- **결정적인 정렬**
- **그 정렬을 그대로 반영한 복합 커서**

정렬이 `ORDER BY created_at DESC, id DESC`라면 커서 조건도 같은 구조를 따라가야 합니다.

Phase 8. 이전 페이지, 다음 페이지, 무한 스크롤은 어떻게 처리할까?

무한 스크롤에서는 보통 "다음 페이지"만 있으면 충분합니다. 이 경우 API는 대개 이런 형태로 설계합니다.

```
{
  "items": [
    { "id": 124, "title": "..." },
    { "id": 123, "title": "..." }
  ],
  "nextCursor": "opaque-token",
  "hasNext": true
}
```

여기서 `nextCursor`는 보통 클라이언트가 내부 구조를 몰라도 되도록 **opaque token**으로 두는 편이 많습니다. 서버는 내부적으로 `createdAt`, `id`, 필터 정보를 인코딩해 저장할 수 있습니다.

`hasNext`는 어떻게 계산할까?

가장 흔한 방법은 **`limit + 1`건을 읽는 것**입니다.

예를 들어 페이지 크기가 20이면:

```
LIMIT 21
```

- 21건이 오면 `hasNext = true`
- 실제 응답에는 앞 20건만 내보냄
- 마지막 1건은 다음 페이지 존재 여부 판단에만 사용

이 방식은 불필요한 `COUNT(*)` 없이도 "다음 페이지가 있는가"를 판단하게 해 줍니다.

이전 페이지는 어떻게 할까?

이전 페이지가 필요하면 보통 **비교 방향과 정렬 방향을 뒤집어 조회한 뒤 애플리케이션에서 다시 뒤집습니다.**

예를 들어 현재 커서보다 앞쪽 데이터를 읽고 싶다면:

```
SELECT id, title, created_at
FROM posts
WHERE status = 'PUBLISHED'
  AND (
    created_at > :cursorCreatedAt
    OR (created_at = :cursorCreatedAt AND id > :cursorId)
  )
ORDER BY created_at ASC, id ASC
LIMIT 20;
```

그다음 애플리케이션에서 결과를 다시 역순으로 뒤집어 원래 정렬(`DESC`)에 맞춥니다.

커서 방식이 불편한 경우

- 사용자가 `37페이지`로 직접 이동해야 함
- 페이지 번호 버튼이 UX 핵심임
- "마지막 페이지" 개념이 중요함

이런 화면은 커서 방식보다 `OFFSET/LIMIT`이 더 자연스럽습니다.

Phase 9. 총 개수(`COUNT`)는 언제 같이 주고 언제 분리할까?

많은 목록 API가 이 응답을 고민합니다.

```
{
  "items": [...],
  "page": 3,
  "size": 20,
  "totalCount": 128731
}
```

문제는 `totalCount`가 공짜가 아니라는 점입니다.

- 필터 조건이 복잡할 수 있습니다
- 조인이나 검색 조건이 들어가면 `COUNT(*)`도 비싸질 수 있습니다

- 데이터가 큰 테이블에서는 목록 조회와 별도로 또 하나의 무거운 쿼리가 추가됩니다

총 개수가 꼭 필요한 경우

- 관리자/백오피스 화면
- 검색 결과 페이지 번호 UI
- 마지막 페이지 계산이 필요한 화면

굳이 필요 없는 경우

- 무한 스크롤
- 피드형 목록
- "더 보기" 버튼 기반 UI
- 높은 QPS의 공개 API

이런 경우에는 `hasNext` 만으로 충분한 경우가 많습니다.

실무에서 자주 하는 선택

1. 목록 조회와 `COUNT` 를 **분리**합니다
2. `COUNT` 가 꼭 필요 없는 화면은 **아예 생략**합니다
3. 약간 오래돼도 되는 개수라면 캐시를 검토합니다

핵심은 이것입니다.

> **목록을 보여 주는 데 꼭 필요하지 않은 총 개수 때문에, 모든 요청에 비싼 집계 쿼리를 얹지 않는다**

Phase 10. 왜 중복과 누락이 생길까?

페이지네이션은 요청이 한 번에 끝나지 않습니다. 첫 페이지를 읽고, 몇 초 뒤 두 번째 페이지를 읽는 동안 **데이터가 바뀔 수 있습니다.**

`OFFSET` 방식에서 생기는 흔한 문제

첫 페이지가 `OFFSET 0 LIMIT 20`이고, 그 사이에 맨 앞에 새 글 1개가 추가됐다고 가정해 보겠습니다.

```
요청 1: 1~20번째 행 조회
새 글 1건이 맨 앞에 삽입
요청 2: OFFSET 20 LIMIT 20
```

그러면 두 번째 요청은 원래 21번째가 아니라, **삽입으로 밀려난 다른 구간**을 읽게 됩니다. 이 과정에서:

- 첫 페이지 마지막 항목이 두 번째 페이지에서 다시 보이거나
- 어떤 항목은 한 번도 안 보이고 건너뛰어질 수 있습니다

삭제가 중간에 일어나도 비슷한 문제가 생깁니다.

커서 방식은 왜 더 안정적일까?

커서 방식은 "이전 페이지 마지막 행 이후"를 기준으로 읽기 때문에, **앞쪽에 새 데이터가 추가되어도 다음 페이지 경계가 덜 흔들립니다.**

하지만 커서도 만능은 아닙니다.

- 정렬 키 자체가 수정되면 레코드 위치가 바뀔 수 있습니다
- 점수 기반 랭킹처럼 정렬 값이 계속 변하면 중복/누락이 여전히 생길 수 있습니다

즉, 커서 방식은 보통 `OFFSET`보다 안정적이지만, **정렬 키가 움직이는 목록에서는 완전한 스냅샷 보장을 해 주지 않습니다.**

정말 같은 스냅샷으로 끝까지 읽어야 한다면

예를 들어 대용량 내보내기(export)처럼 "처음 본 시점의 결과"를 끝까지 유지해야 한다면, 일반적인 HTTP 페이지네이션만으로는 부족할 수 있습니다.

이럴 때는 보통 아래 같은 접근을 검토합니다.

- 첫 페이지 시점의 **상한 정렬 키**를 고정합니다
- 정렬이 `created_at DESC, id DESC`라면 예: `created_at < :maxCreatedAt OR (created_at = :maxCreatedAt AND id ⩽ :maxId)`
- 또는 배치/비동기 작업으로 스냅샷 ID 집합을 따로 만듭니다

즉, **강한 일관성이 필요한 다페이지 조회는 페이지네이션 방식만의 문제가 아니라 스냅샷 설계 문제**가 됩니다.

Phase 11. 언제 `OFFSET`, 언제 커서를 쓰면 좋을까?

실무 기준으로 압축하면 아래 표에 가깝습니다.

상황	추천 방식	이유
게시판형 목록 + 페이지 번호 이동	`OFFSET/LIMIT`	사용자가 임의 페이지로 이동하기 쉽습니다
관리자 테이블 + 총 건수 표시	`OFFSET/LIMIT`	`page`, `totalCount`, `lastPage` 개념과 잘 맞습니다
피드, 타임라인, 무한 스크롤	커서 기반	깊은 페이지와 데이터 삽입에 더 안정적입니다
채팅방 목록, 알림 목록, 활동 로그	커서 기반	최신순 이어 읽기가 자연스럽습니다
대용량 목록을 끝까지 순차 탐색	커서 기반	깊은 `OFFSET` 비용을 피하기 쉽습니다

실무에서 흔한 하이브리드

완전히 하나만 쓰는 것이 아니라, 아래처럼 섞는 경우도 많습니다.

- 사용자 화면 피드: 커서 기반
- 관리자 검색 화면: `OFFSET/LIMIT`

- 첫 몇 페이지는 `OFFSET`, 깊은 탐색 API는 별도 커서 제공

중요한 것은 "유행하는 방식"이 아니라 **화면이 요구하는 탐색 방식과 데이터 특성**입니다.

Phase 12. 실무 체크리스트

페이지네이션 API를 만들 때는 아래 항목을 먼저 확인하면 좋습니다.

- `ORDER BY`가 결정적인가? 예: `created_at DESC, id DESC`
- 정렬과 필터를 받쳐 줄 복합 인덱스가 있는가?
- 깊은 페이지 탐색이 중요한가, 아니면 페이지 번호 이동이 중요한가?
- 페이지 크기 상한을 두고 있는가? 예: 최대 100
- 커서 컬럼이 잘 안 바뀌는 값인가?
- `tie-breaker` 없이 단일 컬럼 커서를 쓰고 있지 않은가?
- `hasNext`만으로 충분한데 매번 `COUNT(*)`를 하고 있지 않은가?
- 실행 계획으로 실제 인덱스 사용 여부를 확인했는가?
- 데이터 삽입/삭제가 발생하는 상황에서 중복·누락 테스트를 해 봤는가?

한눈에 보는 선택 기준

지금까지의 차이를 실무 관점에서 줄이면 아래 표에 가깝습니다.

기준	`OFFSET/LIMIT`	커서 기반
구현 난이도	더 단순합니다	커서 인코딩과 정렬 키 설계가 더 필요합니다
페이지 번호 이동	잘 맞습니다	잘 안 맞습니다
무한 스크롤	가능하지만 깊은 페이지에서 불리합니다	가장 잘 맞습니다
깊은 페이지 성능	뒤로 갈수록 불리해집니다	상대적으로 더 안정적입니다
총 개수 표시	같이 주기 쉽습니다	보통 `hasNext` 중심으로 갑니다
데이터 삽입 시 경계 안정성	앞쪽 데이터 변동에 더 취약합니다	보통 더 안정적입니다

정리

1. **페이지네이션의 출발점은 `LIMIT`이 아니라 `ORDER BY`입니다** — `tie-breaker`까지 포함한 결정적인 정렬이 먼저 있어야 합니다
2. **`OFFSET/LIMIT`은 단순하고 여전히 유용합니다** — 관리자 화면, 페이지 번호 이동, 총 개수 표시가 중요한 UI와 잘 맞습니다
3. **커서 기반은 깊은 페이지와 무한 스크롤에 더 잘 맞습니다** — 앞 구간을 반복해서 버리지 않아 대용량 최신순 목록에서 유리합니다
4. **성능은 페이지네이션 문법보다 인덱스 설계에 더 크게 좌우됩니다** — 필터, 정렬, `tie-breaker`를 함께 보고 `EXPLAIN`으로 확인해야 합니다

5. **중복·누락 문제는 데이터 변경과 함께 봐야 합니다** — 목록이 자주 바뀌는 환경일수록 경계 안정성과 스냅샷 요구사항을 분리해서 판단해야 합니다

핵심을 한 문장으로 줄이면 이렇습니다.

> **페이지네이션은 `LIMIT` 문법의 문제가 아니라, 정렬과 인덱스와 일관성과 UX를 함께 맞추는 설계 문제입니다**