9. 인덱스2

#0.강의/2.데이터베이스로드맵/2.기본

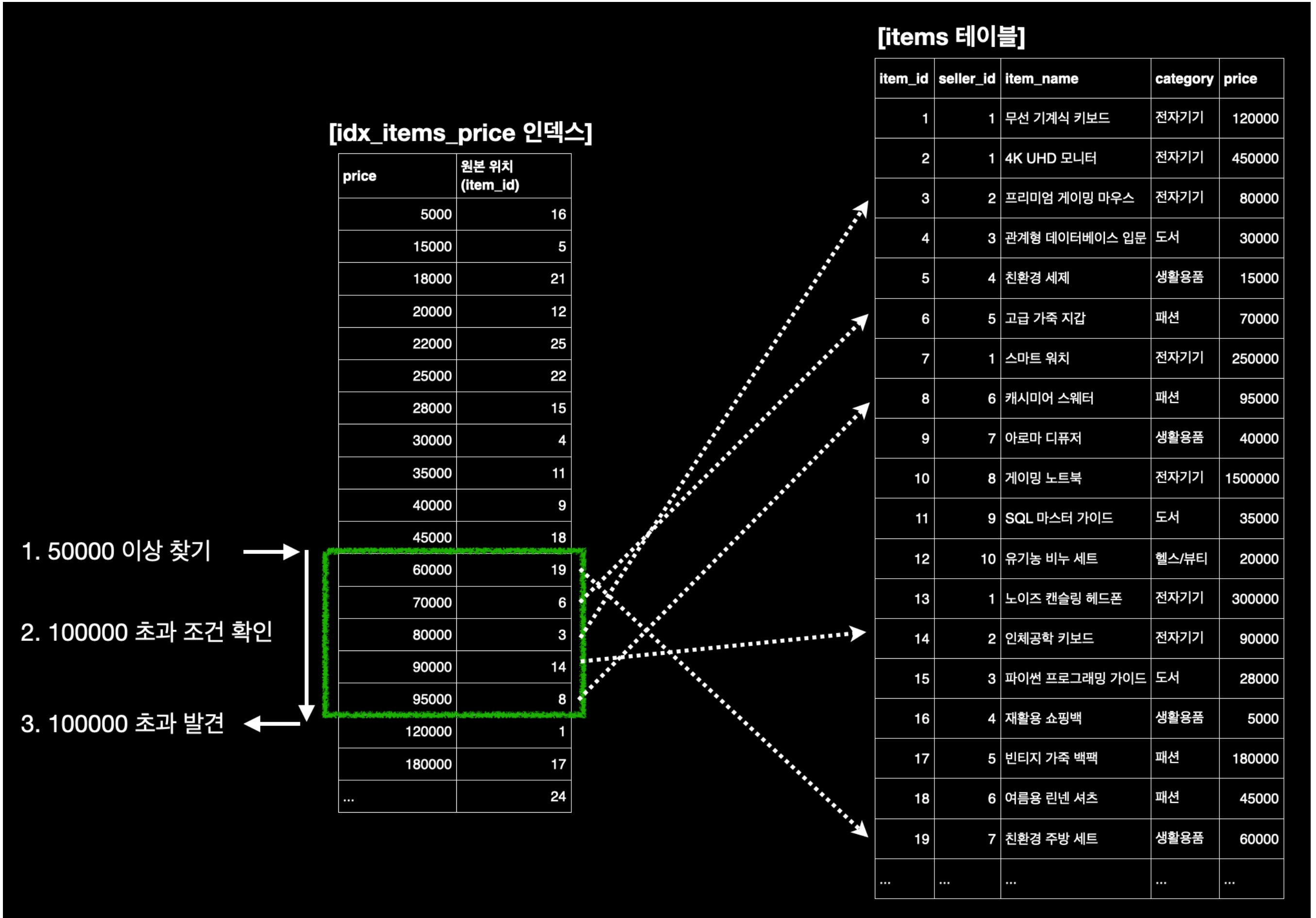
- /옵티마이저와 인덱스 선택
- /커버링 인덱스
- /복합 인덱스1
- /복합 인덱스2
- /복합 인덱스3
- /복합 인덱스 정리
- /인덱스 설계 가이드라인
- /인덱스의 단점과 주의사항
- /문제와 풀이
- /정리

옵티마이저와 인덱스 선택

컬럼에 인덱스를 생성하면, 해당 컬럼을 조건으로 사용하는 모든 `WHERE` 절의 성능이 향상될 것이라고 기대하기 쉽다. 하지만 항상 그렇지는 않다. 데이터베이스의 **옵티마이저(Optimizer)**는 쿼리를 실행하기 전에 여러 실행 가능한 방법을 평가하고, 그중 가장 비용이 적게 드는, 즉 가장 효율적이라고 판단되는 방법을 선택한다.

이 과정에서 옵티마이저는 **인덱스를 사용하는 것이 오히려 비효율적이라고 판단**하면, 인덱스가 존재하더라도 과감히 포기하고 **테이블 전체를 스캔(Full Table Scan)**하는 방법을 선택할 수 있다.

[인덱스 사용 예시]



price	원본 위치 (item_id)
5000	16
15000	5
18000	21
20000	12
22000	25
25000	22
28000	15
30000	4
35000	11
40000	9
45000	18
60000	19
70000	6
80000	3
90000	14
95000	8
120000	1
180000	17
...	24

item_id	seller_id	item_name	category	price
1	1	무선 기계식 키보드	전자기기	120000
2	1	4K UHD 모니터	전자기기	450000
3	2	프리미엄 게이밍 마우스	전자기기	80000
4	3	관계형 데이터베이스 입문	도서	30000
5	4	친환경 세제	생활용품	15000
6	5	고급 가죽 지갑	패션	70000
7	1	스마트 워치	전자기기	250000
8	6	캐시미어 스웨터	패션	95000
9	7	아로마 디퓨저	생활용품	40000
10	8	게이밍 노트북	전자기기	1500000
11	9	SQL 마스터 가이드	도서	35000
12	10	유기농 비누 세트	헬스/뷰티	20000
13	1	노이즈 캔슬링 헤드폰	전자기기	300000
14	2	인체공학 키보드	전자기기	90000
15	3	파이썬 프로그래밍 가이드	도서	28000
16	4	재활용 쇼핑백	생활용품	5000
17	5	빈티지 가죽 백팩	패션	180000
18	6	여름용 린넨 셔츠	패션	45000
19	7	친환경 주방 세트	생활용품	60000
...	...	...	...	...

- 인덱스를 사용하면 검색 대상의 양은 줄어들지만, `items` 테이블의 여러 위치에 흩어진 데이터에 **랜덤하게 접근**해야 한다. 예를 들어, 19, 6, 3, 14, 8번 행 순서로 데이터를 읽어오는 식이다.
- 풀 테이블 스캔을 사용하면 `items` 테이블을 순서대로 쭉 읽으면 된다.

인덱스 손익분기점

옵티마이저가 인덱스 사용 여부를 결정하는 핵심 기준은 바로 **'손익분기점'**이다. 여기서 손익분기점이란, 인덱스를 통해 데이터를 읽는 비용이 테이블 전체를 직접 읽는 비용보다 높아지는 지점을 의미한다.

- **인덱스를 사용하는 비용**: 인덱스 탐색 비용 + 인덱스에서 찾은 주소로 테이블에 접근하는 비용 (랜덤 I/O)
- **풀 테이블 스캔 비용**: 테이블 전체를 순차적으로 읽는 비용 (순차 I/O)

일반적으로 **전체 데이터의 약 20~25% 이상을 조회해야 하는 쿼리**는 인덱스를 통해 테이블의 각 행에 개별적으로 접근하는 것보다, 차라리 테이블 전체를 순차적으로 스캔하는 것이 더 효율적이라고 알려져 있다. 즉, 조회하려는 데이터의 양이 손익분기점을 넘어가면 옵티마이저는 인덱스 사용을 포기한다.

왜 랜덤 I/O가 더 느릴까?

랜덤 I/O가 순차 I/O보다 느린 이유는 데이터를 찾는 과정에서 발생하는 시간 때문이다. 이를 HDD, SSD 디스크를 예로 들어 책 읽기에 비유해보자.

- **순차 I/O (Sequential I/O) → 책을 1페이지부터 순서대로 읽기**
  - 데이터가 저장된 위치를 한 번 찾으면, 그 이후로는 순서대로 쭉 읽기만 하면 된다.
  - HDD의 경우 디스크의 헤드가 거의 움직이지 않고 연속된 데이터를 읽어오므로 작업이 매우 빠르고 효율적이다.
  - SSD의 경우 "여기서부터 100개 읽어와" 라는 하나의 큰 명령으로 처리할 수 있다.

- **랜덤 I/O (Random I/O) → 책의 여러 페이지를 순서 없이 찾아 읽기**
  - 5페이지를 읽은 후 200페이지를 읽고, 다시 45페이지를 읽는 것과 같다.
  - HDD의 경우 읽어야 할 데이터가 디스크의 여러 곳에 흩어져 있어, 데이터를 읽을 때마다 디스크 헤드가 물리적으로 해당 위치까지 이동해야 한다. 이렇게 **데이터의 위치를 찾는 데 걸리는 시간(탐색 시간, Seek Time)이 추가**되기 때문에 순차 I/O에 비해 느리다.
  - SSD의 경우 찾아야 하는 데이터가 100개라면 "이거 읽어와", "저거 읽어와" 라는 100개의 작은 명령을 각각 처리해야 한다. 작은 명령을 여러 번 처리하는 것은 SSD 컨트롤러에 더 많은 오버헤드(부하)를 준다.

데이터베이스에서 인덱스를 통해 테이블의 여러 행에 접근하는 것은, 인덱스에 저장된 주소에 따라 디스크의 여러 위치를 오가야 하는 **랜덤 I/O**를 유발할 수 있다. 반면, 테이블 전체를 스캔하는 것은 처음부터 끝까지 데이터를 읽는 **순차 I/O**에 해당한다.

이 때문에 조회할 데이터가 아주 많을 경우, 여러 번의 랜덤 I/O를 수행하는 것보다 한 번의 순차 I/O가 더 빠를 수 있다.

이제 실제 예시를 통해 이 개념을 확인해 보자.

예시 1: 인덱스를 사용하는 효율적인 범위 검색

먼저, 옵티마이저가 인덱스를 사용하는 것이 효율적이라고 판단하는 경우다. `items` 테이블의 `price` 컬럼에는 `idx_items_price` 인덱스가 생성되어 있다.

```
EXPLAIN SELECT * FROM items WHERE price BETWEEN 50000 AND 100000;
```

[실행 결과]

id	type	possible_keys	key	rows	filtered	Extra
1	range	idx_items_price	idx_items_price	5	100.00	Using index condition

- `type: range`: 옵티마이저는 `idx_items_price` 인덱스를 사용해 특정 범위만 스캔했다.
- `rows: 5`: 조회할 데이터가 5건으로 예상된다. 이는 전체 25건 중 20%에 해당하므로, 손익분기점을 넘지 않는다.
- `key: idx_items_price`: 따라서 옵티마이저는 `idx_items_price` 인덱스를 사용하는 효율적인 실행 계획을 세웠다.

예시 2: 인덱스를 포기하는 비효율적인 범위 검색

이번에는 `WHERE` 절의 범위를 훨씬 더 넓게 잡아보자.

기존에 `50000 ~ 100000` ➜ `1000 ~ 200000`으로 검색 범위를 확 넓혔다.

```
EXPLAIN SELECT * FROM items WHERE price BETWEEN 1000 AND 200000;
```

[실행 결과]

id	type	possible_keys	key	rows	filtered	Extra
1	ALL	idx_items_price	NULL	25	76.00	Using where

실행 계획이 완전히 달라졌다.

- `possible_keys: idx_items_price`: 옵티마이저는 `idx_items_price` 인덱스를 사용할 수 있다는 것을 **알고 있었다.** `possible_keys`는 이 쿼리에서 사용할 수 있는 인덱스 후보이다.
- `key: NULL`: 하지만 최종적으로 인덱스를 **사용하지 않기로 결정했다.**
- `type: ALL`: 결국 선택된 방법은 **풀 테이블 스캔**이다.
- `filtered: 76.00`: 옵티마이저는 이 쿼리가 전체 데이터(25건)의 약 76%, 즉 19건 정도를 반환할 것이라고 예측했다. 이 정도면 손익분기점을 훌쩍 넘는 수치다.

옵티마이저는 19건의 데이터를 찾기 위해 인덱스를 읽고, 다시 테이블에 19번의 개별적인 접근(Random I/O)을 하는 것보다, 그냥 테이블 전체(25건)를 한 번에 쭉 읽는 것(Sequential I/O)이 더 저렴하다고 판단한 것이다.

이처럼 **인덱스는 만능이 아니다.** `WHERE` 절에 인덱스가 걸린 컬럼을 사용하더라도, 조회하려는 데이터의 범위가 너무 넓어 손익분기점을 넘어가면 옵티마이저는 인덱스를 사용하지 않을 수 있다. 따라서 쿼리 튜닝을 할 때는 `EXPLAIN`을

통해 옵티마이저가 실제로 인덱스를 사용하고 있는지 반드시 확인하는 습관을 들여야 한다.

> **여러 인덱스가 있다면?**
> 선택할 수 있는 인덱스 후보가 여러 개 있다면 옵티마이저는 그 중에서 가장 효율적으로 작동하는 인덱스를 선택한다. 물론 이 경우에도 풀 테이블 스캔이 가장 효율적이라고 판단하면 풀 테이블 스캔을 선택할 수 있다.

데이터가 많이 부족하다면?

데이터 자체가 많이 부족하다면 옵티마이저는 **풀 테이블 스캔을 선택할 가능성이 있다.**

테이블에 데이터가 몇 건 없다면, 테이블 전체를 순차적으로 읽는 비용이 인덱스를 탐색하고 테이블에 접근하는 비용보다 훨씬 저렴하기 때문이다.

1,000페이지짜리 두꺼운 책에서는 색인(인덱스)을 보고 원하는 페이지를 찾아가는 것이 빠르다. 하지만 단 3페이지짜리 얇은 소책자에서 특정 내용을 찾을 때는, 굳이 색인을 볼 필요 없이 그냥 1페이지부터 빠르게 훑어보는 것이 더 효율적이다.

마찬가지로 옵티마이저도 테이블이 몇 페이지 되지 않을 정도로 작다면, 굳이 인덱스를 사용하는 복잡한 과정을 거치지 않고 테이블 전체를 직접 스캔하는 것이 더 효율적이라고 판단한다.

이는 개발 환경에서 자주 발생하는 오해 중 하나다. 개발 중인 테이블에 소량의 테스트 데이터만 넣고 쿼리를 실행했을 때, `EXPLAIN` 결과에 `type: ALL`이 표시되어 '인덱스가 왜 작동하지 않지?'라고 생각할 수 있다. 하지만 이는 옵티마이저의 지극히 합리적인 판단일 가능성이 높다. 프로덕션 환경에서 데이터가 수만, 수백만 건으로 늘어나면, 옵티마이저는 다시 인덱스를 사용하는 효율적인 실행 계획을 선택하게 될 것이다.

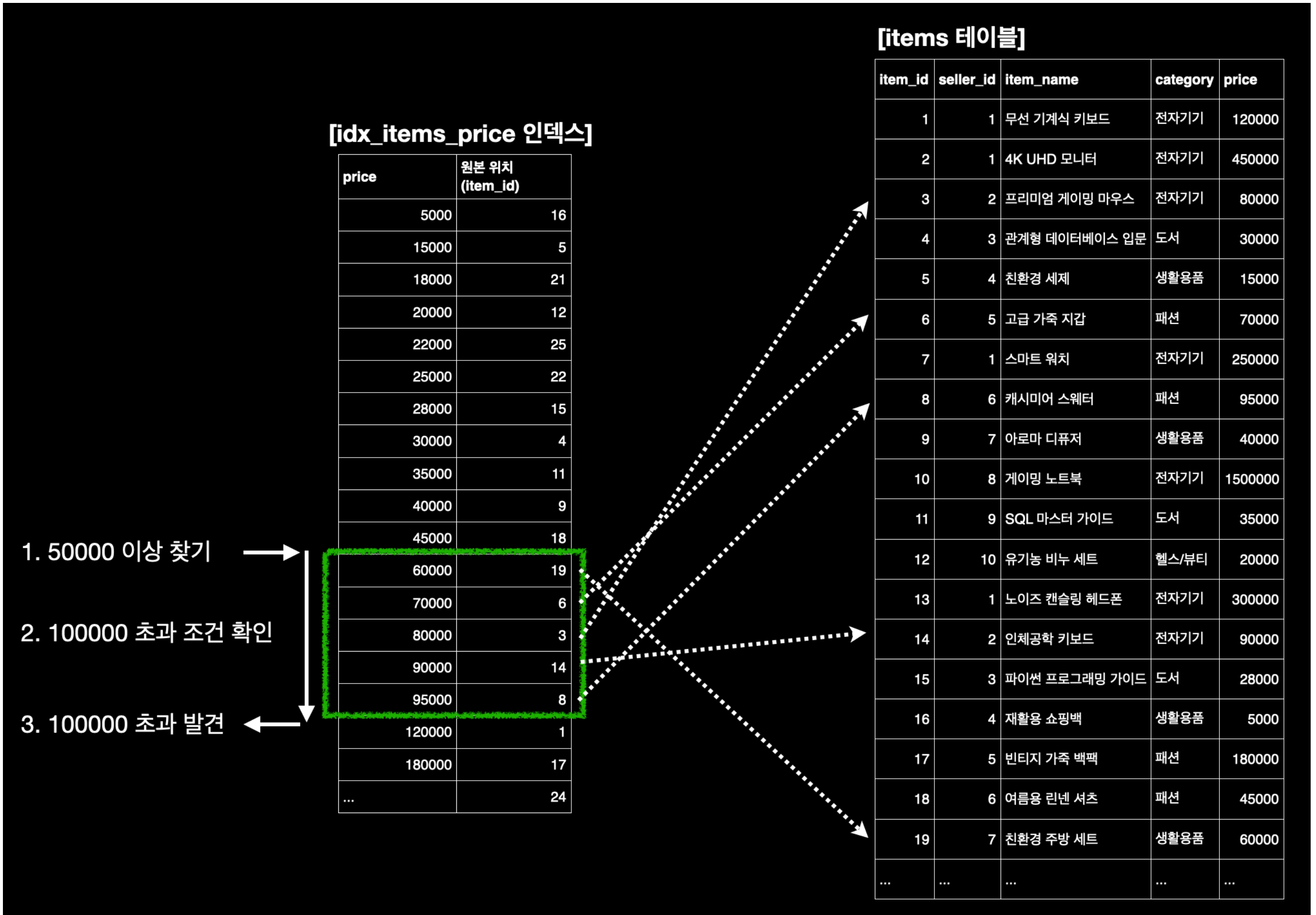
참고로 테스트 목적으로 인덱스를 강제로 적용하려면 다음과 같이 FORCE INDEX를 사용하면 된다.

```sql
SELECT * FROM my_table FORCE INDEX (idx_my_index) WHERE column = 'value';
```

이 방법을 사용하면 쿼리 옵티마이저가 최적의 인덱스를 선택할 수 없기 때문에 실무에서는 권장하지 않는다. 꼭 필요하다면 주의해서 사용해야 한다.

커버링 인덱스

이전 시간에 우리는 옵티마이저가 인덱스를 사용하기로 결정해도, 추가적인 작업이 필요하다는 것을 보았다. 프로세스를 다시 한번 복기해 보자.



price	원본 위치 (item_id)
5000	16
15000	5
18000	21
20000	12
22000	25
25000	22
28000	15
30000	4
35000	11
40000	9
45000	18
60000	19
70000	6
80000	3
90000	14
95000	8
120000	1
180000	17
...	24

item_id	seller_id	item_name	category	price
1	1	무선 기계식 키보드	전자기기	120000
2	1	4K UHD 모니터	전자기기	450000
3	2	프리미엄 게이밍 마우스	전자기기	80000
4	3	관계형 데이터베이스 입문	도서	30000
5	4	친환경 세제	생활용품	15000
6	5	고급 가죽 지갑	패션	70000
7	1	스마트 워치	전자기기	250000
8	6	캐시미어 스웨터	패션	95000
9	7	아로마 디퓨저	생활용품	40000
10	8	게이밍 노트북	전자기기	1500000
11	9	SQL 마스터 가이드	도서	35000
12	10	유기농 비누 세트	헬스/뷰티	20000
13	1	노이즈 캔슬링 헤드폰	전자기기	300000
14	2	인체공학 키보드	전자기기	90000
15	3	파이썬 프로그래밍 가이드	도서	28000
16	4	재활용 쇼핑백	생활용품	5000
17	5	빈티지 가죽 백팩	패션	180000
18	6	여름용 린넨 셔츠	패션	45000
19	7	친환경 주방 세트	생활용품	60000
...	...	...	...	...

1. **인덱스 스캔**: `idx_items_price` 인덱스에서 `WHERE` 조건에 맞는 데이터(의 위치)를 찾는다.
2. **테이블 데이터 접근**: 인덱스에서 찾은 위치 정보(PK인 `item_id`)를 사용해, 원본 `items` 테이블에 접근해서 `SELECT` 절에서 요구하는 다른 컬럼(`*` 또는 `item_name` 등)의 데이터를 가져온다.

이 과정은 마치 책의 '찾아보기'에서 원하는 키워드와 페이지 번호를 찾은 뒤(1단계), 다시 그 페이지로 직접 넘어가서 내용을 읽는 것(2단계)과 같다.

여기서 2단계, 즉 **원본 테이블에 다시 접근하는 과정은 랜덤 I/O를 유발**하므로 비용이 발생한다. 그렇다면 테이블에 접근하는 이 두 번째 단계를 아예 생략할 방법은 없을까? 쿼리에 필요한 모든 데이터를 인덱스에서만 읽어서 끝낼 수는 없을까?

이 질문에 대한 해답이 바로 **커버링 인덱스(Covering Index)**다.

커버링 인덱스란?

커버링 인덱스는 **쿼리에 필요한 모든 컬럼을 포함하고 있는 인덱스**를 말한다. '커버링'이라는 이름 그대로, 인덱스 하나가 쿼리의 요구사항 전체를 '덮는다'는 의미다.

데이터베이스 옵티마이저는 쿼리를 실행할 때, 만약 특정 인덱스가 `SELECT`, `WHERE`, `ORDER BY`, `GROUP BY` 절에 사용되는 모든 컬럼을 가지고 있다면, **원본 테이블에 전혀 접근하지 않고 오직 인덱스만을 읽어서 쿼리를 처리**한다.

이는 디스크의 여러 곳을 오가는 비싼 랜덤 I/O 작업을 완전히 제거하고, 순차 I/O에 가까운 인덱스 스캔만으로 쿼리를 끝낼 수 있음을 의미한다. 당연히 성능은 비약적으로 향상된다.

실제 예시를 통해 커버링 인덱스의 강력함을 확인해 보자.

예시: 커버링 인덱스의 적용 전과 후

쇼핑몰에서 가격이 50,000원에서 100,000원 사이인 상품들의 **가격(`price`)과 이름(`item_name`)**을 조회하는, 매우 흔한 쿼리가 있다고 가정하자.

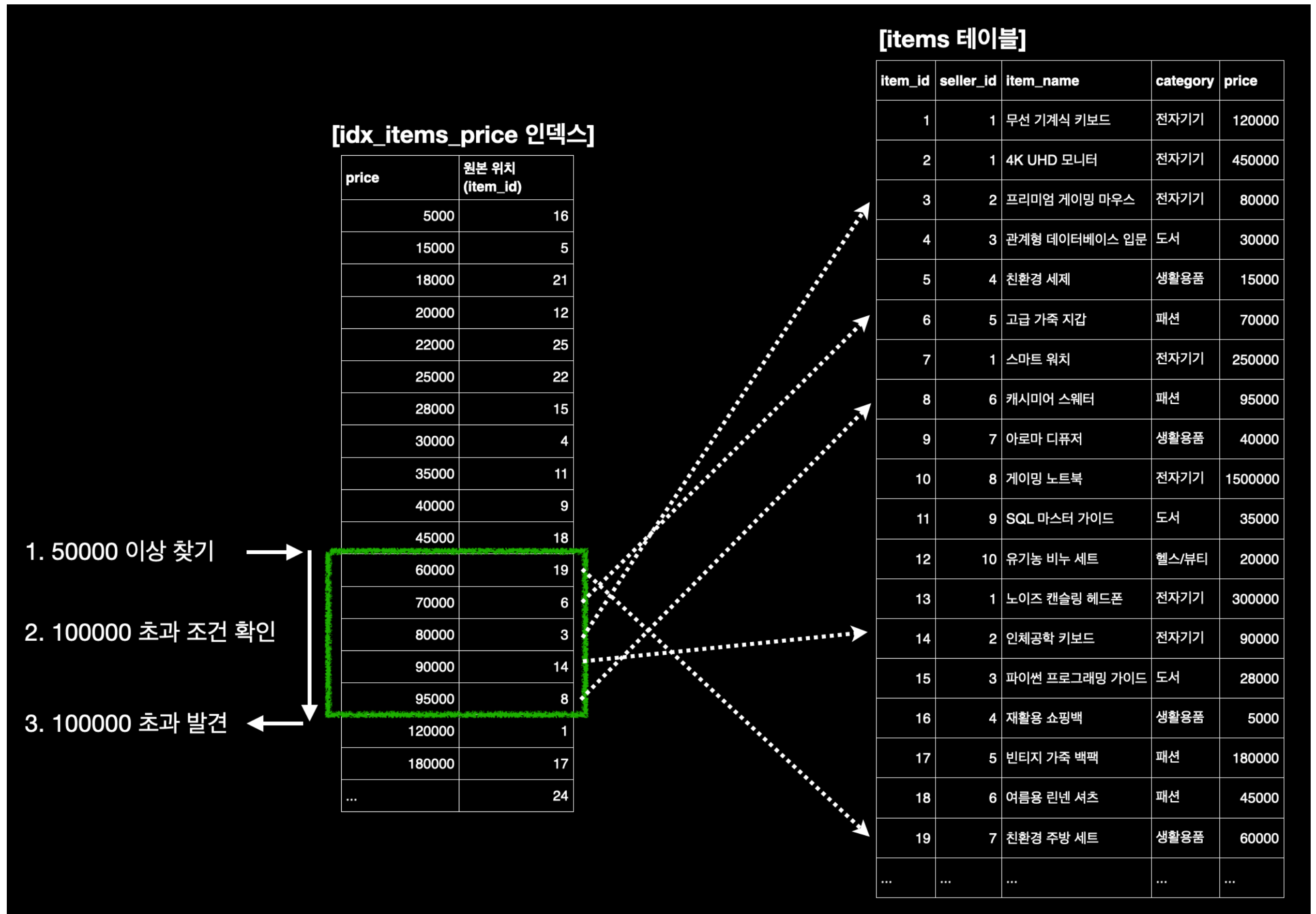
```
SELECT item_id, price, item_name FROM items WHERE price BETWEEN 50000 AND
100000;
```

[실행 결과]

item_id	price	item_name
19	60000	친환경 주방 세트
6	70000	고급 가죽 지갑
3	80000	프리미엄 게이밍 마우스
14	90000	인체공학 키보드
8	95000	캐시미어 스웨터

커버링 인덱스 적용 전 (일반 인덱스 사용)

현재 `items` 테이블에는 `price` 컬럼에 `idx_items_price` 인덱스가 걸려 있다. 이 상태에서 위 쿼리의 실행 계획을 확인해 보자.




```sql
EXPLAIN SELECT item_id, price, item_name FROM items WHERE price BETWEEN 50000
AND 100000;
```

[실행 결과]

id	type	key	rows	filtered	Extra
1	range	idx_items_price	5	100.00	Using index condition

실행 계획을 분석해 보자.

- `key: idx_items_price`: `price` 조건을 위해 `idx_items_price` 인덱스를 사용했다.
- `Extra: Using index condition`: `WHERE` 조건절을 필터링하는 데 인덱스를 효율적으로 사용했지만, 최종 데이터를 가져오기 위해서는 추가 작업이 필요하다는 의미를 내포한다. 이 쿼리에서는 `SELECT` 절의

`item_name` 컬럼이 `idx_items_price` 인덱스에 포함되어 있지 않다. 따라서 옵티마이저는 다음과 같이 동작한다.

1. `idx_items_price` 인덱스를 스캔하여 `price`가 조건에 맞는 행의 `item_id`를 5개 찾는다.
2. 찾아낸 5개의 `item_id`를 사용해, `items` 테이블의 원본 데이터에 5번 접근하여 각각의 `item_name`을 가져온다. (5번의 랜덤 I/O 발생)

이처럼 인덱스에 포함되지 않은 컬럼(`item_name`)을 조회해야 하므로, 테이블 접근을 피할 수 없다.
참고로 MySQL의 인덱스는 테이블의 기본 키(PK, `item_id`)를 기본으로 포함한다.

따라서 `idx_items_price` 인덱스를 사용하는 경우 `item_id`, `price` 두 컬럼의 값은 인덱스에서 바로 조회할 수 있다.

커버링 인덱스 적용 - 인덱스 컬럼만 조회하는 경우

앞서 `idx_items_price` 인덱스는 `price` 컬럼과 기본 키인 `item_id`를 포함하고 있다고 설명했다. 그렇다면 `SELECT` 절에서 `item_name`을 제외하고 `item_id`와 `price`만 조회한다면 어떻게 될까?

이 경우, 쿼리에 필요한 모든 컬럼(`item_id`, `price`)이 `idx_items_price` 인덱스에 이미 포함되어 있으므로, 이 인덱스는 **커버링 인덱스**의 역할을 수행할 수 있다.

실제 쿼리를 실행하면 다음과 같이 `item_id`와 `price`만 조회된다.

```sql
SELECT item_id, price FROM items WHERE price BETWEEN 50000 AND 100000;
```

[실행 결과]

item_id	price
19	60000
6	70000
3	80000
14	90000
8	95000

[커버링 인덱스 사용 그림]

[idx_items_price 인덱스]

price	원본 위치 (item_id)
5000	16
15000	5
18000	21
20000	12
22000	25
25000	22
28000	15
30000	4
35000	11
40000	9
45000	18
60000	19
70000	6
80000	3
90000	14
95000	8
120000	1
180000	17
...	24

1. 50000 이상 찾기

2. 100000 초과 조건 확인

3. 100000 초과 발견

items 테이블 접근X

[items 테이블]

item_id	seller_id	item_name	category	price
1	1	무선 기계식 키보드	전자기기	120000
2	1	4K UHD 모니터	전자기기	450000
3	2	프리미엄 게이밍 마우스	전자기기	80000
4	3	관계형 데이터베이스 입문	도서	30000
5	4	친환경 세제	생활용품	15000
6	5	고급 가죽 지갑	패션	70000
7	1	스마트 워치	전자기기	250000
8	6	캐시미어 스웨터	패션	95000
9	7	아로마 디퓨저	생활용품	40000
10	8	게이밍 노트북	전자기기	1500000
11	9	SQL 마스터 가이드	도서	35000
12	10	유기농 비누 세트	헬스/뷰티	20000
13	1	노이즈 캔슬링 헤드폰	전자기기	300000
14	2	인체공학 키보드	전자기기	90000
15	3	파이썬 프로그래밍 가이드	도서	28000
16	4	재활용 쇼핑백	생활용품	5000
17	5	빈티지 가죽 백팩	패션	180000
18	6	여름용 린넨 셔츠	패션	45000
19	7	친환경 주방 세트	생활용품	60000
...	...	...	...	...

- 쿼리에 사용하는 `item_id`, `price` 컬럼이 `idx_items_price` 인덱스에 모두 있다.
- 따라서 `items` 원본 테이블에 접근할 필요가 없다.
- 결과적으로 `items` 원본 테이블에 접근하지 않고, `idx_items_price` 인덱스만 사용한다.

`item_id`와 `price`만 조회하는 쿼리의 실행 계획을 직접 확인해 보자.

```
EXPLAIN SELECT item_id, price FROM items WHERE price BETWEEN 50000 AND 100000;
```

[실행 결과]

id	type	key	rows	filtered	Extra
1	range	idx_items_price	5	100.00	Using where; Using index

`Extra` 컬럼에 `Using index`가 추가된 것을 확인할 수 있다.

- `Extra: Using index`: 이 표시가 가장 중요하다. 이것은 쿼리에 필요한 모든 데이터를 **오직 인덱스에서만 읽어서 처리했음**을 의미한다. 옵티마이저는 `idx_items_price` 인덱스만 스캔하여 `price`와 `item_id`를 모

두 얻었고, 원본 `items` 테이블에는 전혀 접근할 필요가 없었다.

- `Extra: Using where`: `Using index`와 함께 `Using where`가 표시되는 것을 볼 수 있다. 이는 인덱스 내에서 `WHERE price BETWEEN ...` 조건절을 사용해 불필요한 데이터를 필터링했음을 의미한다. `Using index`와 함께 사용되는 `Using where`는 테이블에 접근한 후 필터링하는 것이 아니라, **인덱스 스캔 단계에서 효율적으로 필터링**이 이루어졌음을 나타낸다.

결론적으로, 이 실행 계획은 **커버링 인덱스를 활용해 테이블 접근을 피했고(**`Using index`**)**, 인덱스 내에서 `WHERE` 절의 조건으로 필터링(`Using where`)을 수행한, 매우 효율적인 쿼리임을 보여준다.

이처럼 `SELECT` 절에 어떤 컬럼을 지정하느냐에 따라, 같은 인덱스라도 커버링 인덱스로 동작할 수도 있고 아닐 수도 있다.
하지만 우리가 원래 원했던 것은 `item_name`까지 함께 조회하는 것이었다. 이처럼 인덱스에 포함되지 않은 컬럼까지 조회하려면 어떻게 해야 할까? 바로 이때, 쿼리에 필요한 컬럼을 모두 포함하는 새로운 인덱스를 직접 만들어주어야 한다.

커버링 인덱스 적용 - item_name 추가

이제 커버링 인덱스를 만들어 `item_name`을 포함하는 쿼리의 성능을 최적화해 보자. 이 쿼리에 필요한 컬럼은 `WHERE` 절의 `price`와 `SELECT` 절의 `item_name`이다. 따라서 이 두 컬럼을 모두 포함하는 인덱스를 생성하면 된다.

> **컬럼이 여러 개인 복합 인덱스에서 컬럼의 순서는 매우 중요하다. `WHERE` 절에서 동등 비교나 범위 검색에 사용되는 컬럼을 가장 앞에 두어야 인덱스를 효율적으로 사용할 수 있다. 여기서는 `price`를 먼저 두고, 그 다음에 `item_name`을 둔다.**
> 복합 인덱스에 대한 부분은 뒤에서 다시 설명한다.

```sql
-- 기존 price 인덱스는 삭제하고, price와 item_name을 포함하는 새로운 복합 인덱스를 생성한다.
DROP INDEX idx_items_price ON items;
CREATE INDEX idx_items_price_name ON items (price, item_name);
```

새로운 `idx_items_price_name` 인덱스는 `price`로 먼저 정렬되고, `price`가 같다면 `item_name`으로 다시 정렬된 구조를 가진다. 이제 이 인덱스는 쿼리에 필요한 `price`와 `item_name` 정보를 모두 가지고 있다.

이 상태에서 다시 한번 동일한 쿼리의 실행 계획을 확인해 보자.

```
EXPLAIN SELECT item_id, price, item_name FROM items WHERE price BETWEEN 50000
AND 100000;
```

[커버링 인덱스 사용 - 그림]

[실행 결과]

id	type	key	rows	filtered	Extra
1	range	idx_items_price_name	5	100.00	Using where; Using index

`Using index` 를 통해 인덱스 만으로 쿼리가 실행되는 것을 확인할 수 있다.

이 쿼리는 이제 `idx_items_price_name` 인덱스만 읽고 끝나므로, 테이블 접근으로 인한 랜덤 I/O가 완전히 사라져 훨씬 빠르고 효율적으로 동작한다.

실제 쿼리를 실행해 보면 결과는 이전과 동일하다. 하지만 그 결과를 얻어오는 내부 과정의 효율성은 매우 큰 차이가 난다.

```sql
SELECT item_id, price, item_name FROM items WHERE price BETWEEN 50000 AND 100000;
```

[실행 결과]

item_id	price	item_name
19	60000	친환경 주방 세트
6	70000	고급 가죽 지갑
3	80000	프리미엄 게이밍 마우스
14	90000	인체공학 키보드
8	95000	캐시미어 스웨터

커버링 인덱스의 장단점

- **장점**
  - **압도적인 `SELECT` 성능 향상**: 테이블 접근을 위한 랜덤 I/O를 제거하여 조회 성능을 극적으로 개선한다.
  - **특히 `COUNT` 쿼리 최적화**: `SELECT COUNT(*)` 와 같은 쿼리에서 테이블 전체가 아닌, 크기가 훨씬 작은 인덱스만 스캔하여 결과를 빠르게 반환할 수 있다.
- **단점**
  - **저장 공간 증가**: 인덱스는 원본 데이터와 별도의 저장 공간을 차지한다. 인덱스에 포함되는 컬럼이 많아질수록 인덱스의 크기도 커진다.
  - **쓰기 성능 저하**: `INSERT`, `UPDATE`, `DELETE` 작업 시, 테이블 데이터뿐만 아니라 인덱스도 함께 수정해야 한다. 인덱스가 많고 복잡할수록 쓰기 작업에 대한 부하가 커진다.

언제 사용해야 할까?

커버링 인덱스는 만능 해결책이 아니며, **읽기 성능과 쓰기 성능 사이의 트레이드오프(trade-off)**를 신중하게 고려해야

한다.

- **조회(읽기)가 매우 빈번하고, 쓰기 작업은 상대적으로 적은 테이블**에 적용하는 것이 가장 효과적이다.
- `SELECT` 절에서 조회하는 컬럼의 개수가 적을 때 유리하다. `SELECT *` 처럼 모든 컬럼을 조회하는 쿼리는 커버링 인덱스의 이점을 누리기 어렵다. (모든 컬럼을 포함하는 인덱스를 만들 수는 있지만, 이는 사실상 테이블을 복제하는 것과 같아 매우 비효율적이다.)
- 성능 저하가 발생하는 특정 쿼리를 튜닝하기 위한 '비장의 무기'로 사용하는 경우가 많다.

복합 인덱스1

지금까지 우리는 하나의 컬럼으로 구성된 **단일 인덱스(Single-column Index)**에 대해 주로 알아보았다. 하지만 실제 쇼핑몰 운영 환경에서는 여러 조건을 조합해서 데이터를 검색하는 경우가 훨씬 더 많다.

예를 들어, "카테고리가 '전자기기'인 상품들 중에 가격이 100,000원 이상인 상품을 보여줘"와 같은 요구사항은 매우 흔하다.

```sql
SELECT * FROM items
WHERE category = '전자기기' AND price >= 100000
```

이런 다중 조건 쿼리의 성능을 최적화하기 위해 사용하는 것이 바로 **복합 인덱스(Composite Index)** 또는 **다중 컬럼 인덱스(Multi-column Index)**다.

복합 인덱스는 이름 그대로 **두 개 이상의 컬럼을 묶어서 하나의 인덱스로 만드는 것**이다.
커버링 인덱스 강의에서 `(price, item_name)` 인덱스를 만들었던 것을 기억하는가? 그것이 바로 복합 인덱스다.

하지만 복합 인덱스를 제대로 사용하려면 한 가지 매우 중요한 규칙을 이해해야 한다. 바로 **'컬럼의 순서'**다. 인덱스를 어떤 컬럼 순서로 만드느냐에 따라 쿼리 성능이 하늘과 땅 차이로 달라질 수 있다.

왜 컬럼 순서가 중요할까?

복합 인덱스의 동작 원리는 우리가 실생활에서 사용하는 '전화번호부'나 '국어사전'과 똑같다.

- **전화번호부**: '성(Last Name)'으로 먼저 정렬된 후, 같은 성 안에서 '이름(First Name)'으로 다시 정렬된다.
- **국어사전**: '첫 번째 글자'로 먼저 정렬된 후, 같은 첫 글자로 시작하는 단어들끼리 '두 번째 글자'로 다시 정렬된다.

`items` 테이블에 `(category, price)` 순서로 복합 인덱스를 만들었다고 상상해 보자. 이 인덱스는 내부적으로 다음과 같이 정렬된다.

1. `category` **를 기준으로 먼저 정렬**한다. ('도서', '생활용품', '전자기기', '패션', '헬스/뷰티' 순서)
2. **같은** `category` **내에서는** `price` **를 기준으로 다시 정렬**한다.

[idx_items_category_price 인덱스 예시]

category	price	item_id
도서	18000	21
도서	22000	25
도서	28000	15
도서	30000	4
도서	35000	11
생활용품	5000	16
생활용품	15000	5
생활용품	40000	9
생활용품	60000	19
전자기기	80000	3
전자기기	90000	14
전자기기	120000	1
전자기기	200000	24
전자기기	250000	7
전자기기	300000	13
전자기기	350000	23
전자기기	450000	2
전자기기	800000	20

전자기기	1500000	10
패션	45000	18
패션	70000	6
패션	95000	8
패션	180000	17
헬스/뷰티	20000	12
헬스/뷰티	25000	22

이 구조를 보면 왜 컬럼 순서가 중요한지 감이 올 것이다.

예를 들어보자.

- `category` **로 검색**: 매우 효율적이다. 인덱스의 앞부분만 보고 빠르게 찾을 수 있다. (예: '전자기기' 섹션으로 바로 점프)
- `category` **와** `price` **로 검색**: 역시 매우 효율적이다. '전자기기' 섹션을 찾은 뒤, 그 안에서 `price` 순으로 정렬된 데이터를 탐색하면 된다.
    - '전자기기' 안에서는 `price` 가 항상 정렬된 상태를 유지한다.
    - 각각의 카테고리 안에서는 `price` 가 항상 정렬된 상태를 유지한다.
    - 1차 정렬의 기준 안에서 2차 정렬은 항상 정렬된 상태를 유지한다.
- `price` **만으로 검색**: **매우 비효율적이다.** 전화번호부에서 성은 모르고 이름 만으로 찾는 것과 같다. `price` 값은 각 `category` 섹션마다 흩어져 있기 때문에, 인덱스 전체를 다 훑어봐야 한다. 이런 경우 옵티마이저는 차라리 풀 테이블 스캔을 선택할 수도 있다.

`price` 만으로 검색하면 왜 인덱스를 사용하기 어려운지 자세히 알아보자.
`idx_items_category_price` 인덱스 예시에서 `price` 컬럼만 딱 분류해서 보자. 가격순으로 정렬이 되어 있는 것 같아 보이지만 중간에 정렬이 다시 틀어진다. 결과적으로 `price` 컬럼만 보면 정렬이 되어 있지 않다. 결과적으로 1차 정렬의 기준 없이 2차 정렬의 값만 가지고는 정렬된 상태를 유지할 수 없다.

[idx_items_category_price 인덱스에서 price만 확인 예시]

price	item_id
18000	21
22000	25

28000	15
30000	4
35000	11
5000	16
15000	5
40000	9
60000	19
80000	3
90000	14
120000	1
200000	24
250000	7
300000	13
350000	23
450000	2
800000	20
1500000	10
45000	18
70000	6
95000	8
180000	17
20000	12
25000	22

이처럼 복합 인덱스는 **첫 번째 컬럼을 기준으로 정렬된 상태에서만 제 역할을 할 수 있다.** 이를 **인덱스 왼쪽 접두어 규칙 (Index Left-Prefix Rule)**이라고 한다. 인덱스를 `(A, B, C)` 순서로 생성했다면, `WHERE` 조건에 다음과 같이 사용될 때 효율적이다.

- `(A)`
- `(A, B)`
- `(A, B, C)`

하지만 `(B)`, `(C)`, `(B, C)` 와 같이 첫 번째 기준인 `A`가 빠진 조건으로는 인덱스를 제대로 활용할 수 없다.

> **복합 인덱스 대원칙**
>
> 복합 인덱스를 설계하고 사용할 때는 다음 세 가지 대원칙을 반드시 기억하자!
>
> 1. **인덱스는 순서대로 사용하라!** (왼쪽 접두어 규칙)
> 2. **등호(=) 조건은 앞으로, 범위 조건(<, >)은 뒤로!**
> 3. **정렬(ORDER BY)도 인덱스 순서를 따르라!**

복합 인덱스의 대원칙을 순서대로 하나씩 배워보자.

복합 인덱스 준비 과정

복합 인덱스를 실습하기 위해 반드시! 기존에 존재하는 `idx`로 시작하는 모든 인덱스를 제거하자
강의를 복습하는 과정에서 지금 과정보다 이후에 만들어진 인덱스가 존재할 수도 있다. 찾아서 모두 제거하자.

```
SHOW INDEX FROM items;
```

- 실행 결과 `idx`로 시작하는 모든 인덱스를 제거하자

인덱스는 아쉽게도 `IF EXISTS` 구문이 없다. 하나하나를 직접 제거해야 한다.

```
DROP INDEX idx_items_item_name ON items;
DROP INDEX idx_items_price_name ON items;
DROP INDEX idx_items_price ON items;
DROP INDEX idx_items_price_category_temp ON items;
DROP INDEX idx_items_category_price ON items;
```

- 만약 복습이라면 추가된 인덱스가 더 있을 수 있다. 이전에 만들어진 `idx`로 시작하는 인덱스는 모두 제거해야 한다.

실습을 위해 `(category, price)` 순서로 복합 인덱스를 생성하자.

```
CREATE INDEX idx_items_category_price ON items (category, price);
```

만들어진 인덱스를 최종 확인하자.

```
SHOW INDEX FROM items;
```

결과는 반드시 다음과 같이 `idx` 로 시작하는 인덱스가 `idx_items_category_price` 하나만 추가되어야 한다.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Cardinality
items	0	PRIMARY	1	item_id	25
items	1	fk_items_sellers	1	seller_id	10
items	1	idx_items_category_price	1	category	5
items	1	idx_items_category_price	2	price	25

- `idx_items_category_price` 는 하나의 복합 인덱스이다.
- `Seq_in_index` 는 복합 인덱스의 컬럼 순서를 나타낸다. `idx_items_category_price` 의 경우 1번은 `category`, 2번은 `price` 순서로 만들어진 것을 확인할 수 있다.

> ⓘ 주의
> 복합 인덱스를 실습하기 위해 반드시! 기존에 존재하는 `idx` 로 시작하는 모든 인덱스를 제거하자
> `idx` 로 시작하는 인덱스는 `idx_items_category_price` 인덱스만 존재해야 한다.

복합 인덱스 성공 예제1: `category` 사용

복합 인덱스의 첫 번째 컬럼만 `WHERE` 절에 사용하는 경우다. 이는 **인덱스 왼쪽 접두어 규칙**을 가장 잘 활용하는 기본

적인 예시다.

"카테고리가 '전자기기'인 모든 상품을 찾아보자."

이제 이 인덱스가 있는 상태에서 쿼리의 실행 계획을 확인하자

```
EXPLAIN SELECT * FROM items WHERE category = '전자기기';
```

[idx_items_category_price 인덱스]

category	price	item_id (원본 참조)
도서	18000	21
도서	22000	25
도서	28000	15
도서	30000	4
도서	35000	11
생활용품	5000	16
생활용품	15000	5
생활용품	40000	9
생활용품	60000	19
전자기기	80000	3
전자기기	90000	14
전자기기	120000	1
전자기기	200000	24
전자기기	250000	7
전자기기	300000	13
전자기기	350000	23
전자기기	450000	2
전자기기	800000	20
전자기기	1500000	10
패션	45000	18
...	...	...

[실행 결과]

id	type	key	rows	filtered	Extra
1	ref	idx_items_category_price	10	100.00	NULL

실행 계획을 분석해 보자.

- `type: ref`: `category`가 '전자기기'인 조건을 만족하는 데이터를 찾기 위해 동등 비교(`=`)나 `JOIN`을 사용하고, 인덱스를 효율적으로 사용했음을 의미한다. 쉽게 이야기해서 Key 값 하나를 딱 집어서 찾은 것이다.
- `key: idx_items_category_price`: 우리가 만든 복합 인덱스가 사용되었다.
- `rows: 10`: 옵티마이저는 '전자기기' 카테고리에 해당하는 상품이 약 10건 있을 것으로 정확히 예측하고, 그 부분만 탐색한다.

이는 `(category, price)`로 정렬된 인덱스에서 `category`가 '전자기기'인 첫 번째 위치를 빠르게 찾아낸 뒤, '전자기기'가 끝날 때까지 인덱스를 순차적으로 읽기만 하면 되므로 매우 효율적이다. 전화번호부에서 '김'씨를 찾는 것과 같이, 'ㄱ' 섹션으로 바로 이동해서 '김'씨가 끝날 때까지 읽는 것과 같다.

복합 인덱스 성공 예제2: `category`, `price` 함께 사용

이번에는 복합 인덱스를 구성하는 모든 컬럼을 `WHERE` 절에 사용하는 경우다. 인덱스의 필터링 능력을 최대로 활용하는 상황이다.

"카테고리가 '전자기기'이면서, 가격이 정확히 120,000원인 상품을 찾아보자."

```
EXPLAIN SELECT * FROM items WHERE category = '전자기기' AND price = 120000;
```

[idx_items_category_price 인덱스]

category	price	item_id (원본 참조)
도서	18000	21
도서	22000	25
도서	28000	15
도서	30000	4
도서	35000	11
생활용품	5000	16
생활용품	15000	5
생활용품	40000	9
생활용품	60000	19
전자기기	80000	3
전자기기	90000	14
전자기기	120000	1
전자기기	200000	24
전자기기	250000	7
전자기기	300000	13
전자기기	350000	23
전자기기	450000	2
전자기기	800000	20
전자기기	1500000	10
패션	45000	18
...	...	...

[실행 결과]

id	type	key	rows	filtered	Extra
1	ref	idx_items_category_price	1	100.00	NULL

실행 계획은 훨씬 더 효율적으로 보인다.

- `type: ref` : 두 개의 컬럼 조건(`category`, `price`)을 모두 만족하는 데이터를 찾기 위해 인덱스를 사용했다.
- `rows: 1` : 탐색할 행의 수가 단 `1` 개로 예측된다.

데이터베이스는 먼저 `category` 가 '전자기기'인 섹션을 찾고, 그 안에서 `price` 가 `120000` 인 지점을 탐색한다. '전자기기' 섹션 내부는 이미 `price` 순으로 정렬되어 있으므로, 원하는 데이터를 매우 빠르게 특정할 수 있다.

이는 전화번호부에서 '김수로'라는 사람을 찾는 것과 같다. '김'씨 섹션을 찾고, 그 안에서 '수로'라는 이름을 찾아내는 과정은 매우 빠르다. 이처럼 인덱스를 구성하는 모든 컬럼을 조건으로 사용하면 가장 효과적으로 데이터를 필터링할 수 있다.

복합 인덱스 성공 예제3: 복합 인덱스와 정렬

복합 인덱스의 진정한 힘은 **정렬(`ORDER BY`) 작업을 피할 때** 드러난다. `WHERE` 절의 필터링과 `ORDER BY` 의 정렬 방향이 인덱스 순서와 일치하면, 데이터베이스는 불필요한 `filesort` 작업을 생략해서 성능을 크게 향상시킬 수 있다.

"카테고리가 '전자기기'이면서 100,000원 초과인 상품을 가격 오름차순으로 정렬해서 보여줘." 라는 요구사항을 생각해보자.

```
EXPLAIN SELECT * FROM items WHERE category = '전자기기' AND price > 100000
ORDER BY price;
```

[idx_items_category_price 인덱스]

category	price	item_id (원본 참조)
도서	18000	21
도서	22000	25
도서	28000	15
도서	30000	4
도서	35000	11
생활용품	5000	16
생활용품	15000	5
생활용품	40000	9
생활용품	60000	19
전자기기	80000	3
전자기기	90000	14
전자기기	120000	1
전자기기	200000	24
전자기기	250000	7
전자기기	300000	13
전자기기	350000	23
전자기기	450000	2
전자기기	800000	20
전자기기	1500000	10
패션	45000	18
...	...	...

[실행 결과]

id	type	key	rows	filtered	Extra
1	range	idx_items_category_price	8	100.00	Using index condition

가장 주목해야 할 부분은 `Extra` 컬럼에 `Using filesort` **가 없다는 점**이다. 이것이 어떻게 가능할까?

데이터베이스의 동작 과정을 따라가 보자.

1. `idx_items_category_price` 인덱스를 사용해 `category` 가 '전자기기'인 섹션으로 빠르게 이동한다.
2. 해당 섹션 내에서, `price` 가 `100000` 을 초과하는 첫 번째 데이터를 찾는다.
3. 그 지점부터 '전자기기' 섹션이 끝날 때까지 인덱스를 **순서대로 읽기만 하면 된다.**

왜냐하면 인덱스의 '전자기기' 섹션은 이미 `price` **순서로 완벽하게 정렬**되어 있기 때문이다. 따라서 데이터베이스는 별도로 데이터를 모아 다시 정렬할 필요 없이, 인덱스를 읽는 즉시 `ORDER BY price` 조건을 만족하는 결과를 얻게 된다.

이처럼 `WHERE` 절의 조건과 `ORDER BY` 절의 조건이 복합 인덱스의 순서(`category` → `price`)와 일치하면, 데이터베이스는 가장 효율적인 방식으로 데이터를 찾고 정렬까지 한 번에 처리한다. `filesort` 를 피하는 것이야말로 복합 인덱스를 사용하는 핵심적인 이유 중 하나다.

쉽게 이야기하면 `ORDER BY` 를 사용할 때 복합 인덱스의 순서대로 정렬하면 추가적인 정렬(`filesort`)을 피할 수 있다는 것이다. 다음 예를 보자. 여기서 복합 인덱스가 `category, price` 순서이므로 `ORDER BY` 도 다음과 같이 이 순서에 맞추어 사용하면 정렬을 최적화 할 수 있다.

```sql
EXPLAIN SELECT * FROM items WHERE category = '전자기기' AND price > 100000
ORDER BY category, price; -- category, price 순서로 정렬
```

그런데 여기서 선택된 `category` 는 '전자기기' 단 하나이므로 이런 경우에는 `category` 를 정렬 조건에 넣을 필요가 없다. 정렬은 최소 2개 이상 있을 때 의미가 있다. 이런 경우에는 `ORDER BY` 에 `category` 를 두는 것이 의미가 없으므로 다음과 같이 생략한다.

```sql
EXPLAIN SELECT * FROM items WHERE category = '전자기기' AND price > 100000
ORDER BY price;
```

당연한 이야기지만 만약 `ORDER BY item_name` 처럼 인덱스 순서와 다른 컬럼으로 정렬을 요청했다면, 조회한 결과를 다시 정렬해야 하기 때문에 Extra 컬럼에 `Using filesort` 가 발생할 것이다.

```
EXPLAIN SELECT * FROM items WHERE category = '전자기기' AND price > 100000
ORDER BY item_name;
```

[실행 결과]

id	type	key	rows	filtered	Extra
1	range	idx_items_category_price	8	100.00	Using index condition; Using filesort

복합 인덱스2

복합 인덱스 실패 예제1: 인덱스 순서 무시

> **복합 인덱스 대원칙**
>
> 복합 인덱스를 설계하고 사용할 때는 다음 세 가지 대원칙을 반드시 기억하자!
>
> 1. **인덱스는 순서대로 사용하라!** (왼쪽 접두어 규칙)
> 2. **등호(=) 조건은 앞으로, 범위 조건(<, >)은 뒤로!**
> 3. **정렬(ORDER BY)도 인덱스 순서를 따르라!**

이제부터 중요한 문제 상황이다. 복합 인덱스의 첫 번째 컬럼(`category`)을 건너뛰고, 두 번째 컬럼(`price`)만으로 데이터를 검색하면 어떻게 될까?

"카테고리와 상관없이 가격이 80,000원인 상품을 찾아보자."

```
EXPLAIN SELECT * FROM items WHERE price = 80000;
```

> ⓘ 주의
> 복합 인덱스를 실습하기 위해 반드시! 기존에 존재하는 `idx` 로 시작하는 모든 인덱스를 제거하자
> `idx` 로 시작하는 인덱스는 `idx_items_category_price` 인덱스만 존재해야 한다.

[실행 결과]

id	type	key	rows	filtered	Extra
1	ALL	NULL	25	10.00	Using where

실행 계획은 처참하다.

- `type: ALL` : **풀 테이블 스캔**이 발생했다.
- `key: NULL` : `idx_items_category_price` 인덱스가 있음에도 불구하고, 옵티마이저는 이 인덱스를 **사용하지 않았다.**

왜 이런 결과가 나왔을까? **인덱스 왼쪽 접두어 규칙** 때문이다.

`idx_items_category_price` 인덱스는 `category` 로 먼저 정렬되어 있다. `price` 가 `80000` 인 상품은 '전자기기' 카테고리에도 있을 수 있고, 만약 있다면 '패션' 카테고리에도 있을 수 있다. 즉, `price` 값은 인덱스 전체에 흩어져 있다.

[idx_items_category_price 인덱스]

category	price	item_id (원본 참조)
도서	18000	21
도서	22000	25
도서	28000	15
도서	30000	4
도서	35000	11
생활용품	5000	16
생활용품	15000	5
생활용품	40000	9
생활용품	60000	19
전자기기	80000	3
전자기기	90000	14
전자기기	120000	1
전자기기	200000	24
전자기기	250000	7
전자기기	300000	13
전자기기	350000	23
전자기기	450000	2
전자기기	800000	20
전자기기	1500000	10
패션	45000	18
...	...	...

데이터베이스 입장에서 `price`만으로 데이터를 찾으려면, '도서' 카테고리 섹션을 처음부터 끝까지 다 보고, '생활용품' 섹션도 다 보고, '전자기기', '패션' 등 모든 카테고리 섹션을 다 뒤져봐야 한다. 이는 인덱스 전체를 스캔하는 작업으로, 차라리 원본 테이블 전체를 스캔하는 것보다 나을 것이 없다.

전화번호부에서 성은 모르고 이름이 '수로'인 사람을 찾으려면, 가나다순으로 모든 페이지를 넘겨봐야 하는 것과 같은 이치다. 이처럼 복합 인덱스는 선행 컬럼 조건 없이는 제 역할을 하지 못한다. 이 규칙을 반드시 기억해야 한다.

복합 인덱스 실패 예제2: 범위 조건을 먼저 사용

> **복합 인덱스 대원칙**
> 복합 인덱스를 설계하고 사용할 때는 다음 세 가지 대원칙을 반드시 기억하자!
> 1. **인덱스는 순서대로 사용하라!** (왼쪽 접두어 규칙)
> 2. **등호(=) 조건은 앞으로, 범위 조건(<, >)은 뒤로!**
> 3. **정렬(ORDER BY)도 인덱스 순서를 따르라!**

복합 인덱스 활용에는 한 가지 더 중요한 제약 조건이 있다. 바로 **선행 컬럼에 범위 조건(`>`, `<`, `BETWEEN`, `LIKE %`) 이 사용되면, 그 뒤에 오는 컬럼은 인덱스를 제대로 활용할 수 없다는 점**이다.

"카테고리명이 '패션' **이상**인 상품들 중에서, 가격이 정확히 20,000원인 상품을 찾아줘." 라는 요구사항을 생각해보자. 문자열 정렬 순서로 보면 카테고리명이 패션 이상인 것은 '패션', '헬스/뷰티'이다.

```sql
EXPLAIN SELECT * FROM items WHERE category >= '패션' AND price = 20000;
```

[실행 결과]

id	type	key	rows	filtered	Extra
1	range	idx_items_category_price	6	10.00	Using index condition

실행 계획을 자세히 분석해 보자. `type`이 `range`이고, `key`에 `idx_items_category_price`가 사용되었으니 언뜻 보기에는 인덱스를 잘 사용한 것처럼 보인다. 하지만 여기서 주목해야 할 것은 `filtered` 컬럼의 값인 `10.00%`와 `Extra` 컬럼의 `Using index condition`이다.

[idx_items_category_price 인덱스]

category	price	item_id (원본 참조)
도서	18000	21
도서	22000	25
도서	28000	15
도서	30000	4
도서	35000	11
생활용품	5000	16
생활용품	15000	5
생활용품	40000	9
생활용품	60000	19
전자기기	80000	3
...	...	...
패션	45000	18
패션	70000	6
패션	95000	8
패션	180000	17
헬스/뷰티	20000	12
헬스/뷰티	25000	22

데이터베이스는 이 쿼리를 어떻게 처리했을까?

1. `idx_items_category_price` 인덱스를 사용해 `category`가 '패션'인 위치를 찾는다. (`>=` 조건의 시작점)
2. 거기서부터 인덱스의 끝까지 **모든 데이터를 스캔**한다. ('패션', '헬스/뷰티' 카테고리에 해당하는 모든 데이터)
3. 스캔하는 각 레코드마다 `price = 20000` 조건을 만족하는지 **하나하나 검사**한다. (인덱스 사용이 아니라 직접 필터링 한다.)

이것이 바로 `filtered: 10.00%`의 의미다. 옵티마이저는 `category >= '패션'` 조건으로 약 6개의 행을 찾을 것으로 예상하고(`rows: 6`), 그 중에서 `price = 20000` 조건을 만족하는 데이터는 10% 정도일 것이라고 예측한다. 즉, `price` 컬럼은 데이터를 효율적으로 '찾는(seek)' 데 사용된 것이 아니라, 일단 `category` 조건으로 넓게 가져온 데이터를 '걸러내는(filter)' 데만 사용된 것이다.

이 부분을 좀 더 쉽게 풀어서 설명하겠다.

데이터베이스는 `category >= '패션'` 조건에 따라 **인덱스**에서 '패션'과 '헬스/뷰티' 섹션을 찾았다. 하지만 그 다음 조건인 `price = 20000`을 처리할 때 문제가 생긴다.

- '패션' 섹션은 `price` 순으로 정렬되어 있다.
- '헬스/뷰티' 섹션도 `price` 순으로 정렬되어 있다.

하지만 '패션' 섹션의 가격들과 '헬스/뷰티' 섹션의 가격들이 **전체적으로 정렬된 것은 아니다.** '패션'의 마지막 상품 가격이 '헬스/뷰티'의 첫 상품 가격보다 높을 수 있다.

패션의 price

price	item_id
45000	18
70000	6
95000	8
180000	17

헬스/뷰티의 price

price	item_id
20000	12
25000	22

이제 처음 조건으로 찾은 `패션, 헬스/뷰티`의 가격을 찾은 순서대로 합쳐서 보자.

패션 + 헬스/뷰티의 price

price	item_id
45000	18
70000	6
95000	8
180000	17
20000	12
25000	22

합쳐진 가격은 이제 정렬된 상태가 아니다! 인덱스를 활용하려면 정렬된 상태여야 한다!
따라서 데이터베이스는 `category >= '패션'` 범위를 스캔하면서 가져온 **각각의 데이터에 대해** `price = 20000` **인지 일일이 확인하는 추가 작업**을 해야만 한다. 이렇게 찾은 **패션 + 헬스/뷰티**의 `price` 컬럼은 정렬되어 있지 않다! 따라서 인덱스를 통한 빠른 탐색이 어렵다.

데이터베이스는 `category >= '패션'` 범위를 인덱스의 `category` 컬럼를 통해 스캔하면서 6개의 행을 빠르게 찾았다. 하지만 가져온 **각각의 데이터에 대해** `price = 20000` **인지 일일이 확인하는 필터링 추가 작업**을 해야만 한다.

이처럼 복합 인덱스에서 **앞선 컬럼(**`category`**)에 범위 조건(**`>=`**)이 걸리는 순간, 데이터베이스는 더 이상 뒤따라오는 컬럼(**`price`**)의 정렬 순서를 활용할 수 없게 된다.** `category`가 '패션'일 때의 `price` 정렬과 `category`가 '헬스/뷰티'일 때의 `price` 정렬은 둘을 합치는 순간 정렬이 깨지기 때문이다.

따라서 데이터베이스는 `category >= '패션'`이라는 범위 조건만 사용해서 인덱스를 스캔하고, 나머지 `price = 20000` 조건은 빠르게 찾지 못하고, 필터링만 수행하는 것이다. 이는 인덱스의 성능을 절반만 활용하는 셈이다.

이처럼 복합 인덱스에서 **어떤 컬럼에 범위 검색을 사용하는 순간, 그 뒤에 오는 컬럼들은 인덱스의 정렬 효과를 제대로 누릴 수 없게 된다.** 따라서 인덱스를 설계할 때는 `=` 조건으로 사용될 컬럼을 범위 조건으로 사용될 컬럼보다 앞에 배치하는 것이 일반적인 최적화 전략이다.

복합 인덱스3

범위 검색은 마지막에 한 번만 사용!

이러한 제약 때문에 복합 인덱스를 설계할 때는 다음과 같은 순서를 따르는 것이 매우 중요하다.

등호(=) 조건을 사용하는 컬럼을 앞에, 범위 조건을 사용하는 컬럼을 뒤에 둔다.

예를 들어, 앞서 사용한 쿼리가 자주 사용된다면,

```
SELECT * FROM items
WHERE category >= '패션' AND price = 20000;
```

최적의 인덱스는 순서를 변경해서 `(price, category)` 순서를 사용해야 한다. 왜냐하면 `price = 20000` 이라는 **등호(=) 조건**을 먼저 처리해서 검색 대상을 크게 줄일 수 있기 때문이다.

이번에는 `price`, `category` 순서의 인덱스를 추가해보자.

```
CREATE INDEX idx_items_price_category_temp ON items (price, category);
```

- 이번 예제에서만 임시로 사용할 예정이다. 따라서 끝에 `temp` 라는 이름을 붙여두었다.

```
EXPLAIN SELECT * FROM items
WHERE category >= '패션' AND price = 20000;
```

[실행 결과]

id	type	possible_keys	key	rows	filtered	Extra
1	range	idx_items_category_price, idx_items_price_category_temp	idx_items_price_category_temp	1	100.00	Using index condition

- 옵티마이저는 `idx_items_category_price`, `idx_items_price_category_temp` 두 인덱스 중에 `idx_items_price_category_temp`가 더 효율적이라고 판단한다.
- `Extra: Using index condition`, `rows: 1`, `filtered: 100.00`을 통해 인덱스를 통해 하나의 데

이터를 찾았고 해당 데이터가 별도의 필터링 없이 바로 선택된 것을 확인할 수 있다.

[idx_items_price_category_temp 인덱스]

price	category	item_id
5000	생활용품	16
15000	생활용품	5
18000	도서	21
20000	헬스/뷰티	12
22000	도서	25
25000	헬스/뷰티	22
28000	도서	15
30000	도서	4
35000	도서	11
40000	생활용품	9
...	...	...

데이터베이스는 `(price, category)` 인덱스를 다음과 같이 활용한다.

1. 인덱스에서 `price` 가 `20000` 인 데이터 블록을 **매우 빠르게 찾는다(lookup)**. 이는 인덱스가 `price` 로 정렬되어 있기 때문에 가능한, 가장 효율적인 `ref` 방식(`=`)의 접근이다.
2. 그렇게 찾아낸 좁은 데이터 집합 안에서 `category >= '패션'` 이라는 **범위 조건**을 만족하는 데이터를 찾는다. 여기서는 '패션', '헬스/뷰티'를 만족하면 된다. `price` 가 `20000` 인 데이터들은 이미 `category` 순으로 정렬되어 있으므로, 이 과정 역시 효율적인 `range` 스캔으로 처리된다.

참고로 이번 예제에서는 같은 가격의 제품이 없으므로 카테고리 순 정렬이 크게 의미는 없다. 같은 가격의 제품이 많이 있다면 의미가 있을 것이다.

이처럼 **가장 변별력 있는 등호(=) 조건을 먼저 처리**해서 작업 범위를 최대한 좁히고, 그 다음에 범위 조건을 처리하는 것이 인덱스 설계의 핵심이다.

만약 순서를 반대로 `(category, price)` 로 했다면, `category >= '패션'` 이라는 범위 검색이 먼저 수행되면서

인덱스의 효율이 떨어졌을 것이다. 이는 '복합 인덱스 실패 예제2'에서 이미 확인했다.

결론적으로, 복합 인덱스를 설계할 때는 어떤 쿼리가 주로 사용될지 예측하고, 해당 쿼리의 `WHERE` 절에 맞게 **'등호 조건 컬럼 → 범위 조건 컬럼'** 순서로 구성하는 것이 성능 최적화의 지름길이다.

기존 인덱스를 잘 활용하자

그런데 이처럼 인덱스를 계속 만드는 것 만이 능사는 아니다. 뒤에서 배우겠지만, 인덱스를 추가하면 그 만큼 관리 비용이 들어간다.
기본적으로 기존에 있는 인덱스를 최대한 잘 활용하고, 그래도 안되면 인덱스 추가를 고려해야 한다.

사실 이번에 사용한 다음 예제는

```
SELECT * FROM items
WHERE category >= '패션' AND price = 20000;
```

이번에 생성한 `idx_items_price_category_temp` 인덱스 없이, 기존에 만든 `idx_items_category_price` 인덱스로도 최적의 성능을 활용할 수 있는 방법이 있다.

이 방법을 알아보기 위해 이번에 만든 `price`, `category` 순서의 인덱스를 먼저 제거하자.

```
DROP INDEX idx_items_price_category_temp ON items;
```

실무 팁: `IN` 절 활용하기

범위 조건 때문에 두 번째 인덱스 컬럼을 활용하지 못하는 이 문제는, `>` 나 `<` 같은 범위 대신 `IN` **절을 사용**함으로써 해결할 수 있는 경우가 많다.

MySQL 옵티마이저는 `IN (...)` 을 하나의 큰 범위로 취급하지 않고, **여러 개의 동등 비교(=) 조건의 묶음**으로 인식하기 때문이다.

`category >= '패션'` 조건은 `category IN ('패션','헬스/뷰티')` 와 논리적으로 동일하다. 이 쿼리를 `IN`을 사용하도록 변경하고 실행 계획을 다시 확인해보자.

범위 검색(>=)을 사용한 기존 검색 조건

```
EXPLAIN SELECT * FROM items WHERE category >= '패션' AND price = 20000;
```

[실행 결과]

id	type	key	rows	filtered	Extra
1	range	idx_items_category_price	6	10.00	Using index condition

- `idx_items_category_price` 인덱스를 사용
- `rows 6` : 패션, 헬스/뷰티를 포함해서 인덱스로 6건의 데이터를 찾을 것으로 예상
- `filtered 10.00%` : 인덱스를 통해 찾은 데이터에서 약 10%를 필터링할 것으로 예상

IN 조건을 사용한 검색 조건

```
EXPLAIN SELECT * FROM items
WHERE category IN ('패션', '헬스/뷰티') AND price = 20000;
```

[실행 결과]

id	type	key	rows	filtered	Extra
1	range	idx_items_category_price	2	100.00	Using index condition

`IN` 연산자를 사용한 실행 결과를 자세히 분석해보자.

범위 연산자(`>=`)를 사용했을 때와 `IN` 연산자를 사용했을 때 `EXPLAIN` 결과가 미묘하게 달라졌다.

- 예상 `rows` 가 6 ➜ 2로 줄어들었다. 인덱스로 찾는 범위가 더 줄어든다는 뜻이다.
- 가장 극적인 변화는 `filtered` 컬럼이 `10.00%` 에서 `100.00%` 로 바뀐 것이다. 이것은 인덱스만을 잘 활용해서 원하는 데이터를 100% 다 찾았다는 의미다. 그래서 인덱스를 통해 찾은 데이터를 100% 다 통과시킨다는 의미이다. 자세히 알아보자.

[idx_items_category_price 인덱스]

category	price	item_id (원본 참조)
도서	18000	21
도서	22000	25
도서	28000	15
도서	30000	4
도서	35000	11
생활용품	5000	16
생활용품	15000	5
생활용품	40000	9
생활용품	60000	19
전자기기	80000	3
...	...	...
패션	45000	18
패션	70000	6
패션	95000	8
패션	180000	17
헬스/뷰티	20000	12
헬스/뷰티	25000	22

`IN` 절을 사용했을 때, MySQL 옵티마이저의 동작 방식은 다음과 같이 바뀐다.

1. 옵티마이저는 `WHERE category IN ('패션', '헬스/뷰티')`를 `WHERE category = '패션' OR category = '헬스/뷰티'` 와 동일하게 인식한다.
2. 따라서 전체 쿼리는 내부적으로 `(category = '패션' AND price = 20000)` 또는 `(category = '헬스/뷰티' AND price = 20000)`를 만족하는 데이터를 찾는 것으로 해석된다.
3. `idx_items_category_price` 인덱스를 사용해 `('패션', 20000)` 조합을 만족하는 데이터를 찾는다. (첫 번째 동등 비교)
4. 이어서 `('헬스/뷰티', 20000)` 조합을 만족하는 데이터를 찾는다. (두 번째 동등 비교)

이렇게 작성한 `IN` 쿼리는 쉽게 비유하자면 다음과 같이 나누어 실행된다.

```
SELECT * FROM items WHERE category = '패션' AND price = 20000;
UNION ALL
SELECT * FROM items WHERE category = '헬스/뷰티' AND price = 20000;
```

- `category` '패션'만 보면 `price`가 완전히 정렬되어 있다. 따라서 `price` 컬럼도 인덱스를 사용해서 원하는 데이터를 빠르게 찾을 수 있다.
- `category` '헬스/뷰티'만 보면 `price`가 완전히 정렬되어 있다. 따라서 `price` 컬럼도 인덱스를 사용해서 원하는 데이터를 빠르게 찾을 수 있다.

핵심은 **범위 검색이 동등 비교(`=`)의 여러 묶음으로 바뀌었다**는 점이다.
`>=` 같은 범위 조건에서는 인덱스의 두 번째 컬럼(`price`)을 제대로 활용할 수 없었지만, `category`와 `price`가 모두 특정 값으로 고정된 동등 비교에서는 **복합 인덱스의 모든 컬럼을 효율적으로 사용**할 수 있다.

즉, 옵티마이저는 `idx_items_category_price` 인덱스를 사용해 `('패션', 20000)` 지점으로 한 번, `('헬스/뷰티', 20000)` 지점으로 또 한 번, 총 두 번의 **정확한 위치 탐색(seek)**을 수행한다. 이 탐색 과정에서 `price` 조건까지 완벽하게 반영되므로, 불필요하게 데이터를 읽고 버리는 과정이 사라진다. 이것이 바로 `filtered` 컬럼이 `100.00%`로 표시되는 이유다.

결론적으로 `>`는 **'연속된 범위'**로 처리되어 복합 인덱스의 추가적인 활용을 막는 반면, `IN`은 '여러 개의 개별 지점'에 대한 동등(`=`) 비교의 묶음으로 처리된다. 옵티마이저는 `IN` 절의 각 값에 대해 인덱스를 사용한 효율적인 탐색(Seek)을 여러 번 수행할 수 있으므로, 복합 인덱스의 모든 컬럼을 효과적으로 활용할 수 있다.

논리적으로 같은 결과를 반환하더라도, `>`는 인덱스 스캔(Scan) 방식으로 동작하여 후속 컬럼 활용에 제한이 있는 반면, `IN`은 여러 번의 탐색(Seek) 방식으로 동작하여 복합 인덱스의 모든 컬럼을 효율적으로 활용할 수 있기 때문에 성능 차이가 발생한다.

물론 `IN` 절에 들어가는 값이 수백, 수천 개로 너무 많아지면 성능이 저하될 수 있지만, 이처럼 범위 조건을 몇 개의 동등 조건으로 바꿀 수 있는 상황이라면 `IN` 절은 매우 강력한 최적화 도구가 될 수 있다.

> 팁
> **실무에서는 범위가 한정적인 컬럼에 이 `IN` 트릭을 자주 사용한다.**
> 예를 들어, 상품 상태를 나타내는 `status` 컬럼이 '판매중', '품절', '판매중지' 3가지 값만 가진다고 하자.
> '판매중' 또는 '품절' 상태인 상품을 찾을 때 `WHERE status >= '판매중'` 과 같이 조회하는 것보다 `WHERE status IN ('판매중', '품절')` 로 조회하는 것이 복합 인덱스를 활용하는 데 훨씬 유리할 수 있다.
> 물론, `IN` 절에 들어가는 값의 개수가 너무 많아지면 오히려 성능이 저하될 수도 있으므로, 항상 `EXPLAIN` 을 통해 실제 실행 계획을 확인하고 결정하는 것이 현명하다.

복합 인덱스 정리

지금까지 복합 인덱스가 성공하는 예제와 실패하는 예제를 다양하게 살펴보았다. `EXPLAIN` 의 결과가 미묘하게 달라지고, 옵티마이저의 동작 방식이 복잡해서 어렵게 느껴질 수도 있다.

하지만 지금까지 배운 모든 내용은 사실 몇 가지 핵심 대원칙으로 귀결된다. 이 원칙들만 제대로 이해하고 있다면, 어떤 복잡한 쿼리를 만나더라도 최적의 인덱스를 설계하고 사용할 수 있게 될 것이다. 실무에서는 이 원칙을 이해하는 것만으로도 많은 성능 문제를 해결할 수 있다.

> **복합 인덱스 대원칙**
> 복합 인덱스를 설계하고 사용할 때는 다음 세 가지 대원칙을 반드시 기억하자!
> 1. **인덱스는 순서대로 사용하라!** (왼쪽 접두어 규칙)
> 2. **등호(=) 조건은 앞으로, 범위 조건(<, >)은 뒤로!**
> 3. **정렬(ORDER BY)도 인덱스 순서를 따르라!**

이 세 가지 원칙이 왜 중요한지 다시 한번 정리해 보자.

1. 인덱스는 순서대로 사용하라!

이는 **'인덱스 왼쪽 접두어 규칙'**을 의미한다. 복합 인덱스는 `(A, B, C)` 순서로 생성되었을 때, `WHERE` 절에서 **A 조

건 없이는 B나 C를 사용할 수 없다. 전화번호부에서 '성'을 모르고 '이름'만으로 사람을 찾을 수 없는 것과 같은 이치다.

- **가능 (`O`)**: `WHERE A`, `WHERE A AND B`, `WHERE A AND B AND C`
- **불가능 (`X`)**: `WHERE B`, `WHERE C`, `WHERE B AND C`

이것이 복합 인덱스의 가장 기본적이고 절대적인 규칙이다.

2. 등호(=) 조건은 앞으로, 범위 조건(<, >)은 뒤로!

복합 인덱스의 컬럼 중 하나에 **범위 조건(`>`, `<`, `BETWEEN`, `LIKE %...`)이 사용되는 순간, 그 뒤에 오는 컬럼은 인덱스의 정렬 효과를 누릴 수 없다.**

예를 들어, `(category, price)` 인덱스가 있을 때 `WHERE category > '도서' AND price = 30000` 쿼리를 생각해보자.

- 데이터베이스는 `category`가 '도서'보다 큰 ('생활용품', '전자기기', ...) 섹션을 인덱스에서 찾는다.
- 하지만 '생활용품' 섹션과 '전자기기' 섹션의 `price`는 서로 연결되어 정렬된 것이 아니다.
- 결국 데이터베이스는 '도서' 이후의 모든 인덱스를 스캔하면서, `price`가 `30000`인지 일일이 확인해야 한다. `price` 컬럼은 필터링에만 사용될 뿐, 탐색(seek)에는 사용되지 못한다.

따라서 가장 효율적인 인덱스 설계는 다음과 같은 순서를 따른다.

1. **변별력이 높은 등호(`=`) 조건**으로 사용할 컬럼을 인덱스 앞쪽에 배치한다. (`IN` 절도 여러 개의 등호 조건으로 취급되어 유리하다.)
2. **범위 조건**으로 사용할 컬럼은 인덱스 뒤쪽에 배치한다.

이렇게 하면 등호 조건으로 검색 대상을 최대한 좁힌 뒤, 그 좁은 범위 내에서만 범위 검색을 수행하므로 성능이 극대화된다.

3. 정렬(ORDER BY)도 인덱스 순서를 따르라!

복합 인덱스의 가장 강력한 기능 중 하나는 **불필요한 정렬 작업을 생략**하게 해주는 것이다. `ORDER BY` 절이 인덱스 컬럼 순서와 일치하면, 데이터베이스는 이미 정렬된 인덱스를 순서대로 읽기만 하면 되므로 매우 빠르다. 이 경우 실행 계획에서 `Using filesort`가 사라지는 것을 볼 수 있다.

- **인덱스**: `(category, price)`
- **빠른 쿼리**: `WHERE category = '전자기기' ORDER BY price`
  - '전자기기' 섹션은 이미 `price` 순으로 정렬되어 있으므로 추가 정렬이 필요 없다.
- **느린 쿼리 (filesort 발생)**: `WHERE category = '전자기기' ORDER BY stock_quantity`
  - 인덱스는 `stock_quantity` 순서와는 무관하므로, 결과를 가져온 뒤 별도로 정렬해야 한다.

`WHERE` 절에서 인덱스를 잘 활용하는 것만큼, `ORDER BY`에서 `filesort`를 피하는 것도 전체 쿼리 성능에 결정적인 영향을 미친다.

결론적으로, 복합 인덱스는 단순히 여러 컬럼을 묶는 것이 아니라, **'순서'와 '조건의 종류(등호/범위)'를 고려한 전략적인 설계**가 핵심이다. 이 세 가지 대원칙만 명심한다면, 여러분도 실무에서 고성능 쿼리를 충분히 잘 작성할 수 있을 것이다.

인덱스 설계 가이드라인

인덱스를 만드는 법(`CREATE INDEX`)을 아는 것보다 더 중요한 것은, **어디에 인덱스를 만들어야 하는지 아는 것**이다. 잘못된 인덱스는 오히려 시스템 성능을 떨어뜨리는 애물단지가 될 수 있다.

인덱스는 결코 공짜가 아니다. 데이터를 추가(`INSERT`), 수정(`UPDATE`), 삭제(`DELETE`)할 때마다 인덱스도 함께 변경되어야 하므로 쓰기 성능이 저하되고, 별도의 저장 공간도 차지한다. 따라서 우리는 이 비용을 상쇄하고도 남을 만큼의 '검색 성능 향상'이라는 이득을 얻을 수 있는 곳에만 전략적으로 인덱스를 생성해야 한다.

이번 시간에는 인덱스 생성의 핵심 전략과 가이드라인에 대해 알아보겠다.

핵심 원칙: 카디널리티 (Cardinality)

인덱스를 어디에 걸지 판단하는 가장 중요한 기준은 바로 **카디널리티(Cardinality)**다.
카디널리티란, 해당 컬럼에 저장된 값들의 **고유성(uniqueness)** 정도를 나타내는 지표다.

- **카디널리티가 높다 (High Cardinality)**: 해당 컬럼에 중복되는 값이 거의 없다는 의미다.
    - 예: `items` 테이블의 `item_id`, `item_name`
- **카디널리티가 낮다 (Low Cardinality)**: 해당 컬럼의 값이 몇 종류 안되어 중복되는 값이 많다는 의미다.
    - 예: `items` 테이블의 `category` (5종류), `is_active` (2종류)

인덱스는 '찾아보기'다. 찾아보기가 효과적이려면, 특정 키워드를 찾았을 때 검색 범위가 확!!! 줄어들어야 한다.
`items` 테이블에서 `WHERE is_active = TRUE` 라는 조건으로 검색한다고 생각해 보자. `is_active` 컬럼에 인덱스가 있더라도, `TRUE`인 데이터가 전체의 80%라면, 데이터베이스는 인덱스를 통해 전체 데이터의 80%를 스캔해야 한다. 이런 경우 데이터베이스 옵티마이저는 "이럴 바엔 그냥 풀 테이블 스캔하는 게 낫겠다"고 판단할 수 있다.
반면 `WHERE item_name = '게이밍 노트북'`은 어떤가? 인덱스는 수십만 건의 상품 데이터 중 단 1건으로 검색 범

위를 완벽하게 좁혀준다.

핵심 규칙: 인덱스는 카디널리티가 높은, 즉 식별력이 좋은 컬럼에 생성할 때 가장 효율적이다.

인덱스 생성 가이드라인

위의 핵심 원칙을 바탕으로, 어떤 컬럼이 인덱스 후보가 되는지 구체적인 가이드라인을 살펴보자.

1. `WHERE` 절에서 자주 사용되는 컬럼

가장 기본적이고 명백한 가이드라인이다. 인덱스의 존재 이유 자체가 `WHERE` 절의 검색 속도를 높이는 것이기 때문이다. 사용자들이 상품을 검색할 때 `items.item_name` 으로 검색하거나, 특정 카테고리(`items.category`)를 필터링한다면 이 컬럼들은 인덱스 생성의 우선 후보가 된다.

2. `JOIN` 의 연결고리가 되는 컬럼 (외래 키)

`JOIN` 의 성능은 연결고리가 되는 컬럼에 인덱스가 있는지 여부에 따라 극적으로 달라진다. '행복쇼핑' 판매자가 등록한 모든 상품을 조회하는 쿼리를 예로 들어보자.

```sql
SELECT
    s.seller_name,
    i.item_name,
    i.price
FROM items i
JOIN sellers s ON i.seller_id = s.seller_id
WHERE s.seller_name = '행복쇼핑';
```

`items.seller_id` 에 인덱스가 없을 때

만약 `items` 테이블의 `seller_id` 컬럼(외래 키)에 인덱스가 없다면, 데이터베이스는 다음과 같이 비효율적으로 동작한다.

1. `sellers` 테이블에서 `seller_name` 이 '행복쇼핑'인 판매자를 찾는다. (seller_id = 1)
2. `items` 테이블의 **모든 행을 처음부터 끝까지 스캔**하면서, `seller_id` 가 `1` 인 상품을 하나씩 찾아낸다.

> **조인의 논리적인 순서와 실제 순서의 차이**
>
> SQL의 논리적인 순서는 조인을 모두 다 한 다음에 WHERE를 실행한다. 하지만 데이터베이스는 최적화를

> 위해 먼저 데이터를 줄인 다음에 조인한다. 이때 최종 결과는 논리적인 순서와 같음을 보장한다.

`items` 테이블에 상품이 100만 개 있다면, `JOIN`을 위해 100만 번의 비교가 일어나는 끔찍한 일이 벌어진다. 풀 테이블 스캔이 발생하는 것이다.

`items.seller_id`에 인덱스가 있을 때
다행히 `items.seller_id`에는 외래 키 제약 조건 덕분에 인덱스가 자동으로 생성되어 있다. 인덱스가 있을 때의 동작은 완전히 다르다.

1. `sellers` 테이블에서 `seller_name`이 '행복쇼핑'인 판매자를 찾는다. (`seller_id = 1`)
2. `items.seller_id` 인덱스를 사용하여 `seller_id`가 `1`인 상품 데이터의 위치를 **곧바로 찾아낸다.** 풀 테이블 스캔이 사라지고 몇 번의 탐색만으로 `JOIN`이 완료된다.

`EXPLAIN`으로 확인해보자.

```sql
EXPLAIN SELECT
    s.seller_name,
    i.item_name,
    i.price
FROM items i
JOIN sellers s ON i.seller_id = s.seller_id
WHERE s.seller_name = '행복쇼핑';
```

[실행 결과]

id	table	type	key	rows	Extra
1	s	const	seller_name	1	Using index
1	i	ref	fk_items_sellers	5	

`items` 테이블(`i`)의 `type`이 `ref`이고, `key`가 `fk_items_sellers`인 것을 볼 수 있다. 이는 `JOIN` 과정에서 `items` 테이블을 조회할 때 `seller_id` 인덱스를 매우 효율적으로 사용했다는 증거다.

따라서 `JOIN`**에 사용되는 외래 키(Foreign Key) 컬럼에는 반드시 인덱스를 생성해야 한다.**

> **MySQL은 외래 키 제약조건을 설정하면 인덱스를 자동으로 생성한다.**
> 종종 외래 키 제약조건을 걸지 않고 데이터베이스를 사용하는 경우도 있다. 이때는 조인 성능을 위해 외래

> 키로 사용되는 컬럼에 반드시 인덱스를 직접 생성해야 한다.

3. `ORDER BY` 절에서 자주 사용되는 컬럼
`ORDER BY`를 사용한 정렬은 데이터의 양이 많을 경우 매우 비용이 큰 작업이다. 데이터베이스는 결과를 반환하기 전에 모든 데이터를 메모리에 올리고 정렬해야 하기 때문이다.

만약 `ORDER BY`에 사용된 컬럼에 인덱스가 있다면 어떨까? B-Tree 인덱스는 이미 데이터가 정렬된 상태로 저장되어 있다. 데이터베이스는 굳이 데이터를 따로 정렬할 필요 없이, 인덱스에 있는 순서 그대로 데이터를 읽기만 하면 된다. 비용이 큰 정렬 작업(filesort)을 완전히 건너뛸 수 있는 것이다.
'최신 등록 상품 목록 10개'를 보여주는 `ORDER BY registered_date DESC LIMIT 10`과 같은 쿼리는 `registered_date` 컬럼에 인덱스가 있을 때 엄청난 성능 향상을 기대할 수 있다.

인덱스의 단점과 주의사항

지금까지 인덱스의 장점, 즉 검색(`SELECT`) 속도를 비약적으로 향상시키는 원리에 대해 배웠다. 이쯤 되면 '그럼 모든 컬럼에 인덱스를 걸면 최고 아닌가?' 라는 순수한 생각을 할 수도 있다.

결론부터 말하자면, 그것은 **데이터베이스 성능을 망치는 최악의 선택**이다.

이번 시간에는 "인덱스는 공짜가 아니다"라는 중요한 명제를 이해하고, 인덱스를 생성하고 유지하는 데 따르는 '비용', 즉 인덱스의 단점과 관리 시의 주의사항에 대해 알아보겠다. 모든 기술에는 명암이 있듯, 인덱스 역시 잘못 사용하면 약이 아니라 독이 될 수 있다.

인덱스는 공짜가 아니다: 인덱스의 단점

인덱스의 단점은 크게 두 가지 비용으로 설명할 수 있다.

1. 저장 공간 (Storage)
인덱스는 원본 테이블과는 별개로, B-Tree 구조를 가진 물리적인 파일로 디스크에 저장된다. 즉, 인덱스를 생성하면 그만큼의 **추가 저장 공간**이 필요하다.

인덱스는 어떻게 구성하는지에 따라 다르지만, 일반적으로 원본 테이블 크기의 약 10% 내외의 공간을 추가로 차지한다고 알려져 있다. 만약 100GB에 달하는 거대한 `items` 테이블이 있고, 여기에 5개의 인덱스를 추가로 생성한다면? 인

덱스만으로 약 50GB라는 무시할 수 없는 추가 디스크 공간이 필요하게 된다. 인덱스를 무분별하게 생성하면 디스크 사용량이 계속해서 늘어나는 것을 보게 될 것이다.

2. 쓰기 성능 (`INSERT`, `UPDATE`, `DELETE`)
이것이 인덱스의 가장 치명적인 단점이자, 반드시 이해해야 할 핵심 트레이드오프다.
인덱스는 `SELECT`의 속도를 높이는 대가로, `INSERT`, `UPDATE`, `DELETE`의 속도를 희생시킨다.

왜 그럴까? 데이터에 변경이 일어날 때마다, 데이터베이스는 원본 테이블뿐만 아니라 이와 관련된 **모든 인덱스를 함께 수정**해야 하기 때문이다.

- `INSERT`: 새로운 상품이 등록되면(`INSERT`), `items` 테이블에 행이 추가된다. 동시에, 이 테이블에 생성된 **모든 인덱스**(예: `PRIMARY`, `seller_id`, `idx_items_category_price`)의 B-Tree에도 새로운 데이터에 대한 키 값과 주소가 추가되어야 한다. 이 과정에서 B-Tree의 정렬 순서를 유지하고 균형을 맞추기 위한 추가적인 연산이 발생한다. 인덱스가 5개라면, 테이블 삽입 1번에 인덱스 삽입 5번의 작업이 추가되는 셈이다.

- `DELETE`: 상품이 삭제되면(`DELETE`), 테이블에서 행이 사라진다. 동시에 **모든 인덱스**에서도 해당 상품에 대한 키 값이 삭제되어야 한다.

- `UPDATE`: 상품 정보가 수정될 때가 가장 복잡하다.
  - 만약 인덱스가 없는 `stock_quantity` 컬럼의 값이 변경된다면, 인덱스는 수정할 필요가 없으므로 비교적 빠르다.
  - 하지만 **인덱스가 있는 `price` 컬럼의 값이 변경된다면**, 데이터베이스는 기존 `price` 값으로 된 인덱스 항목을 '삭제'하고, 새로운 `price` 값으로 인덱스 항목을 '추가'하는 것과 유사한 작업을 수행한다. 왜냐하면 인덱스는 변경된 값에 맞추어 새로운 정렬 상태를 유지해야 하기 때문이다. 이는 `INSERT`와 `DELETE`가 동시에 발생하는 것과 같아 부하가 크다.

실무 가이드: 균형의 미학

이러한 장단점을 고려할 때, 우리는 어떤 전략을 취해야 할까?

워크로드를 분석하라: 읽기 vs 쓰기

- **읽기 중심(Read-heavy) 서비스**: 데이터 분석 시스템, 블로그, 뉴스 사이트처럼 데이터 변경보다는 조회가 훨씬 더 빈번한 서비스라면, 다양한 조회 성능을 높이기 위해 인덱스를 비교적 자유롭게 생성해도 좋다. 우리 쇼핑몰의 상품 조회 기능이 대표적이다.
- **쓰기 중심(Write-heavy) 서비스**: 실시간으로 데이터를 기록하는 로깅 시스템, 주식 거래 시스템, 채팅 서비스처럼 `INSERT`나 `UPDATE`가 매우 빈번한 서비스라면, 인덱스 생성에 매우 신중하고 보수적이어야 한다. 모든 인덱

스는 쓰기 작업에 오버헤드를 추가하기 때문이다. 꼭 필요한 최소한의 인덱스만 유지해야 한다.

"혹시나 해서" 인덱스를 만들지 마라.
사용하지 않는 인덱스는 저장 공간만 차지하고 쓰기 성능만 저하시키는 암적인 존재다. 명확한 목적 없이, "나중에 쓸 것 같아서" 라는 이유로 인덱스를 미리 만드는 것은 좋지 않다. 느린 쿼리가 발견되었을 때, 그 쿼리를 개선하기 위한 목적으로 생성해야 한다.

사용하지 않는 인덱스는 주기적으로 정리하라.
대부분의 데이터베이스는 특정 인덱스가 얼마나 사용되었는지 모니터링하는 기능을 제공한다. 몇 달, 혹은 1년 이상 아무도 사용하지 않는 인덱스가 있다면, 과감하게 삭제하여 시스템 자원을 확보하고 쓰기 성능을 높여야 한다.

인덱스는 `SELECT` 성능을 위한 최고의 무기이지만, 저장 공간과 쓰기 성능이라는 비용을 요구하는 양날의 검과 같다.

지금까지 우리는 데이터를 빠르고 효과적으로 '읽는' 방법에 많은 시간을 투자했다. 하지만 데이터베이스의 더 근본적인 역할은 데이터를 '안전하게 지키는' 것이다.

만약 상품 가격에 음수(-)가 들어가거나, 있지도 않은 판매자 ID로 상품이 등록되는 등 말도 안 되는 '쓰레기 데이터'가 시스템에 저장된다면 어떻게 될까? 이는 분석 결과를 왜곡하고, 심각한 시스템 오류를 야기할 수 있다.

다음 섹션에서는 데이터베이스 스스로가 데이터의 정확성과 일관성을 지키도록 강제하는 최후의 보루, **데이터 무결성**과 **제약 조건**에 대해 알아보겠다.

> ⓘ **인덱스 컬럼은 가공하면 안된다. - 메뉴얼 추가 내용**
> WHERE 절에서 인덱스가 적용된 컬럼을 함수로 감싸거나 계산을 하는 등 가공하게 되면 인덱스가 적용되지 않는다. 이는 실무에서 정말 자주 하는 실수이므로 반드시 기억해야 한다.
>
> 예를 들어 WHERE 절에서 인덱스가 적용된 컬럼에 `SUBSTRING()` 같은 함수를 사용하거나 연산을 하면 **인덱스가 작동하지 않아** 테이블 전체를 스캔하게 되므로 성능이 크게 저하된다.
>
> - **문제:** `WHERE SUBSTRING(item_name, 1, 5) = '게이밍'` 처럼 인덱스 컬럼(`item_name`)을 가공하면, 데이터베이스는 정렬된 인덱스를 활용하지 못하고 모든 데이터를 일일이 확인한다.
>   - `WHERE indexed_column * 10 = 100` 이런 경우도 마찬가지로 인덱스를 사용하지 못한다.
> - **원인:** 인덱스는 가공되지 않은 **원본 값**을 기준으로 만들어지기 때문이다.

- **해결책:** 컬럼 자체를 가공하는 대신, LIKE 연산자를 사용하여 `WHERE item_name LIKE '게이밍%'` 와 같이 조건을 변경해야 인덱스를 효율적으로 사용할 수 있다.

결론: SQL 성능을 높이려면 **인덱스 컬럼은 절대 가공하지 말고 원본 상태 그대로 사용**해야 한다.

문제와 풀이

문제와 풀이를 위한 인덱스 초기화

문제와 풀이 진행 전에 기존에 존재하는 `idx` 로 시작하는 모든 인덱스를 제거하자

강의를 복습하는 과정에서 지금 과정보다 이후에 만들어진 인덱스가 존재할 수도 있다. 찾아서 모두 제거하자.

```
SHOW INDEX FROM items;
```

- 실행 결과 `idx` 로 시작하는 모든 인덱스를 제거하자

인덱스는 아쉽게도 `IF EXISTS` 구문이 없다. 하나하나를 직접 제거해야 한다.

```
DROP INDEX idx_items_item_name ON items;
DROP INDEX idx_items_price_name ON items;
DROP INDEX idx_items_price ON items;
DROP INDEX idx_items_price_category_temp ON items;
DROP INDEX idx_items_category_price ON items;
```

문제: 인덱스들을 만들어서 다음 쿼리 성능을 개선해라.

최근 쇼핑몰의 `items` 테이블에 데이터가 많아지면서, 사용자들이 특정 조건으로 상품을 조회할 때 시스템이 점점 느려진다는 불만이 접수되었다. 원인 파악 결과, 자주 사용되는 조회 쿼리에 인덱스가 걸려있지 않아 전체 데이터를 스캔(Full Table Scan)하고 있었다.

다음은 느리다고 보고된 주요 쿼리이다.

```
SELECT * FROM items
WHERE category = '전자기기' AND is_active = TRUE;
```

```sql
SELECT * FROM items
WHERE category = '전자기기' AND is_active = TRUE
ORDER BY stock_quantity DESC;

SELECT * FROM items
WHERE stock_quantity < 90 AND category = '전자기기' AND is_active = TRUE;

SELECT * FROM items
WHERE stock_quantity < 90 AND category = '전자기기' AND is_active = TRUE
ORDER BY stock_quantity DESC;
```

인덱스를 만들어서 이 쿼리들이 풀 테이블 스캔이 걸리지 않도록 해라.

필요하다면 여러 인덱스를 만들어도 된다.

[정답]

```sql
CREATE INDEX idx_items_category_actvie_stock ON items (category, is_active,
stock_quantity DESC);
```

정리

옵티마이저와 인덱스 선택

- 데이터베이스 옵티마이저는 쿼리 실행 시, 인덱스 사용과 테이블 전체 스캔(Full Table Scan) 중 더 효율적인 방법을 선택한다.
- 인덱스를 사용하는 것이 오히려 비용이 크다고 판단되면(손익분기점 초과), 인덱스가 있어도 사용하지 않는다.
- 일반적으로 전체 데이터의 20~25% 이상을 조회하는 경우, 인덱스를 통한 랜덤 I/O보다 테이블 전체를 순차적으로 읽는 순차 I/O가 더 빠르다고 판단한다.
- 데이터 양이 매우 적을 때도 옵티마이저는 풀 테이블 스캔을 선택할 수 있다.

커버링 인덱스

- 쿼리에 필요한 모든 컬럼을 포함하는 인덱스를 커버링 인덱스라 한다.
- 커버링 인덱스를 사용하면 원본 테이블에 접근하지 않고 인덱스만으로 쿼리를 처리할 수 있어, 랜덤 I/O가 발생하지 않아 성능이 크게 향상된다.
- 실행 계획(`EXPLAIN`)의 `Extra` 컬럼에 `Using index`가 표시되면 커버링 인덱스가 사용된 것이다.
- `SELECT` 성능을 크게 높이지만, 인덱스 크기가 커지고 쓰기(`INSERT`, `UPDATE`, `DELETE`) 성능이 저하되는 단점이 있다.

> **복합 인덱스 대원칙**
>
> 복합 인덱스를 설계하고 사용할 때는 다음 세 가지 대원칙을 반드시 기억하자!
>
> 1. **인덱스는 순서대로 사용하라!** (왼쪽 접두어 규칙)
> 2. **등호(=) 조건은 앞으로, 범위 조건(<, >)은 뒤로!**
> 3. **정렬(ORDER BY)도 인덱스 순서를 따르라!**

복합 인덱스1

- 두 개 이상의 컬럼을 묶어 하나의 인덱스로 만든 것을 복합 인덱스라 한다.
- **컬럼의 순서가 매우 중요하며**, 인덱스는 첫 번째 컬럼부터 순서대로 조건에 사용되어야 한다 (인덱스 왼쪽 접두어 규칙).
- 인덱스가 `(A, B)` 순서라면 `WHERE A=...` 또는 `WHERE A=... AND B=...`는 효율적이지만, `WHERE B=...`는 인덱스를 제대로 활용할 수 없다.
- `WHERE` 절과 `ORDER BY` 절이 인덱스 순서와 일치하면, 불필요한 정렬 작업(`filesort`)을 생략할 수 있어 성능에 매우 유리하다.

복합 인덱스2

- 복합 인덱스의 첫 번째 컬럼을 건너뛰고 두 번째 이후의 컬럼만 조건으로 사용하면 인덱스를 활용할 수 없어 풀 테이블 스캔이 발생할 수 있다.
- 복합 인덱스의 선행 컬럼에 범위 조건(`>`, `<`, `BETWEEN` 등)이 사용되면, 그 뒤에 오는 컬럼은 인덱스의 정렬 효과를 누릴 수 없어 효율이 떨어진다.
- 범위 조건으로 넓게 가져온 데이터를 필터링하는 방식으로 동작하여, 인덱스의 성능을 절반만 활용하게 된다.

복합 인덱스3

- 복합 인덱스 설계 시 **등호(=) 조건을 사용하는 컬럼을 앞에, 범위 조건을 사용하는 컬럼을 뒤에 두는 것**이 일반적인 최적화 전략이다.
- 범위 검색(`>=`) 대신 `IN` **절을 사용하면**, 옵티마이저는 이를 여러 개의 동등 비교(=)로 인식하여 복합 인덱스를 더 효율적으로 활용할 수 있다.

인덱스 설계 가이드라인

- 인덱스 생성의 가장 중요한 기준은 **카디널리티(Cardinality)**, 즉 값의 고유성 정도이다. 중복도가 낮은(카디널리티가 높은) 컬럼에 생성해야 효과적이다.
- 인덱스 생성 가이드라인
  - `WHERE` 절에서 자주 사용되는 컬럼
  - `JOIN` 의 연결고리가 되는 컬럼 (외래 키)
  - `ORDER BY` 절에서 자주 사용되는 컬럼 (정렬 작업 회피)

인덱스의 단점과 주의사항

- 인덱스는 공짜가 아니며, 단점도 명확히 존재한다.
- **저장 공간 차지**: 인덱스는 별도의 파일로 저장되어 추가 디스크 공간을 사용한다.
- **쓰기 성능 저하**: `INSERT`, `UPDATE`, `DELETE` 작업 시 테이블뿐만 아니라 인덱스도 함께 수정해야 하므로 오버헤드가 발생한다. 특히 인덱스 컬럼의 `UPDATE` 는 부하가 크다.
- 읽기 중심 서비스와 쓰기 중심 서비스를 구분하여 필요한 최소한의 인덱스만 생성하고, 사용하지 않는 인덱스는 주기적으로 정리해야 한다.