8. 인덱스1

#0.강의/2.데이터베이스로드맵/2.기본

인덱스를 위한 샘플 데이터

인덱스를 학습하기 위해 새로운 테이블들을 사용하자.

쇼핑몰의 판매자와 상품 정보를 담고 있는 `sellers`와 `items` 테이블이다. 인덱스 예제는 이 두 테이블을 기반으로 한다.

> **SQL 소스 파일 참고**
>
> 강의 자료가 PDF 파일이라 복잡한 SQL 코드를 복사할 때 오류가 발생할 수 있다.
>
> 이 경우, 섹션 1. 강의 소개와 수업 자료 ➜ SQL 소스 파일을 다운로드해서 사용하자.

```
-- 데이터베이스가 존재하지 않으면 생성
CREATE DATABASE IF NOT EXISTS my_shop2;
USE my_shop2;

-- 테이블이 존재하면 삭제 (실습을 위해 초기화)
DROP TABLE IF EXISTS items;
DROP TABLE IF EXISTS sellers;

-- 판매자 테이블 생성
CREATE TABLE sellers (
    seller_id INT PRIMARY KEY AUTO_INCREMENT,
```

```sql
    seller_name VARCHAR(100) UNIQUE NOT NULL,
    registered_date DATE NOT NULL
);

-- 상품 테이블 생성
CREATE TABLE items (
    item_id INT PRIMARY KEY AUTO_INCREMENT,
    seller_id INT NOT NULL,
    item_name VARCHAR(255) NOT NULL,
    category VARCHAR(100) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL,
    registered_date DATE NOT NULL,
    is_active BOOLEAN NOT NULL,

    CONSTRAINT fk_items_sellers FOREIGN KEY (seller_id)
        REFERENCES sellers(seller_id)

);

-- 판매자 데이터 입력
INSERT INTO sellers (seller_id, seller_name, registered_date) VALUES
(1, '행복쇼핑', '2020-01-15'),
(2, '스마트상점', '2021-03-22'),
(3, '글로벌셀러', '2019-11-01'),
(4, '에코마켓', '2022-07-10'),
(5, '베스트딜', '2020-05-30'),
(6, '패션리더', '2023-01-05'),
(7, '리빙스타', '2021-09-12'),
(8, '테크월드', '2022-04-18'),
(9, '북스토리', '2020-08-25'),
(10, '헬스앤뷰티', '2023-03-01');

-- 상품 데이터 입력
INSERT INTO items (item_id, seller_id, item_name, category, price,
stock_quantity, registered_date, is_active) VALUES
(1, 1, '무선 기계식 키보드', '전자기기', 120000, 100, '2022-01-20', TRUE),
(2, 1, '4K UHD 모니터', '전자기기', 450000, 50, '2022-02-15', TRUE),
(3, 2, '프리미엄 게이밍 마우스', '전자기기', 80000, 200, '2021-11-10', TRUE),
(4, 3, '관계형 데이터베이스 입문', '도서', 30000, 500, '2020-05-01', TRUE),
(5, 4, '친환경 세제', '생활용품', 15000, 300, '2023-08-01', FALSE),
(6, 5, '고급 가죽 지갑', '패션', 70000, 120, '2022-06-25', TRUE),
(7, 1, '스마트 워치', '전자기기', 250000, 80, '2023-03-10', TRUE),
```

```
(8, 6, '캐시미어 스웨터', '패션', 95000, 70, '2023-10-05', FALSE),
(9, 7, '아로마 디퓨저', '생활용품', 40000, 150, '2022-09-01', TRUE),
(10, 8, '게이밍 노트북', '전자기기', 1500000, 30, '2023-01-30', TRUE),
(11, 9, 'SQL 마스터 가이드', '도서', 35000, 400, '2021-04-12', TRUE),
(12, 10, '유기농 비누 세트', '헬스/뷰티', 20000, 250, '2023-02-20', FALSE),
(13, 1, '노이즈 캔슬링 헤드폰', '전자기기', 300000, 90, '2023-07-01', TRUE),
(14, 2, '인체공학 키보드', '전자기기', 90000, 110, '2022-05-05', TRUE),
(15, 3, '파이썬 프로그래밍 가이드', '도서', 28000, 600, '2021-01-01', FALSE),
(16, 4, '재활용 쇼핑백', '생활용품', 5000, 1000, '2023-09-15', TRUE),
(17, 5, '빈티지 가죽 백팩', '패션', 180000, 60, '2022-08-01', TRUE),
(18, 6, '여름용 린넨 셔츠', '패션', 45000, 180, '2023-04-20', TRUE),
(19, 7, '친환경 주방 세트', '생활용품', 60000, 130, '2022-10-10', FALSE),
(20, 8, '고성능 그래픽 카드', '전자기기', 800000, 40, '2023-06-01', TRUE),
(21, 9, '어린이를 위한 그림책', '도서', 18000, 700, '2022-03-01', TRUE),
(22, 10, '천연 에센셜 오일', '헬스/뷰티', 25000, 200, '2023-05-10', TRUE),
(23, 1, '휴대용 빔 프로젝터', '전자기기', 350000, 70, '2023-02-01', TRUE),
(24, 2, '게이밍 의자', '전자기기', 200000, 90, '2022-07-20', TRUE),
(25, 3, '세계사 탐험', '도서', 22000, 350, '2021-02-28', FALSE);
```

`sellers` 테이블

`sellers` 테이블은 쇼핑몰에 입점한 **판매자** 정보를 저장한다.

컬럼명	타입	설명
`seller_id`	`INT`	판매자의 고유 식별자 (기본 키)
`seller_name`	`VARCHAR(100)`	판매자의 이름 유니크 제약 조건
`registered_date`	`DATE`	판매자가 쇼핑몰에 등록한 날짜

`items` 테이블

`items` 테이블은 판매자가 등록한 **상품** 정보를 저장한다. 각 상품은 특정 판매자(`seller_id`)에 의해 등록된다.

컬럼명	타입	설명
`item_id`	`INT`	상품의 고유 식별자 (기본 키)
`seller_id`	`INT`	상품을 등록한 판매자의 ID 외래 키(FK)로 `sellers` 테이블의 `seller_id`를 참조

`item_name`	`VARCHAR(255)`	상품의 이름
`category`	`VARCHAR(100)`	상품의 카테고리 (예: '전자기기', '도서', '패션')
`price`	`INT`	상품의 가격
`stock_quantity`	`INT`	상품의 재고 수량
`registered_date`	`DATE`	상품이 등록된 날짜
`is_active`	`BOOLEAN`	상품의 판매 활성화 여부 (TRUE: 활성, FALSE: 비활성)

[셀러 데이터 확인]

```
SELECT * FROM sellers;
```

seller_id	seller_name	registered_date
1	행복쇼핑	2020-01-15
2	스마트상점	2021-03-22
3	글로벌셀러	2019-11-01
4	에코마켓	2022-07-10
5	베스트딜	2020-05-30
6	패션리더	2023-01-05
7	리빙스타	2021-09-12
8	테크월드	2022-04-18
9	북스토리	2020-08-25
10	헬스앤뷰티	2023-03-01

[상품 데이터 확인]

```
SELECT * FROM items;
```

item_id	seller_id	item_name	category	price	registered_date	is_ac
1	1	무선 기계식 키보드	전자기기	120000	2022-01-20	1
2	1	4K UHD 모니터	전자기기	450000	2022-02-15	1
3	2	프리미엄 게이밍 마우스	전자기기	80000	2021-11-10	1
4	3	관계형 데이터베이스 입문	도서	30000	2020-05-01	1
5	4	친환경 세제	생활용품	15000	2023-08-01	0
6	5	고급 가죽 지갑	패션	70000	2022-06-25	1
7	1	스마트 워치	전자기기	250000	2023-03-10	1
8	6	캐시미어 스웨터	패션	95000	2023-10-05	0
9	7	아로마 디퓨저	생활용품	40000	2022-09-01	1
10	8	게이밍 노트북	전자기기	1500000	2023-01-30	1
11	9	SQL 마스터 가이드	도서	35000	2021-04-12	1
12	10	유기농 비누 세트	헬스/뷰티	20000	2023-02-20	0
13	1	노이즈 캔슬링 헤드폰	전자기기	300000	2023-07-01	1
14	2	인체공학 키보드	전자기기	90000	2022-05-05	1
15	3	파이썬 프로그래밍 가이드	도서	28000	2021-01-01	0
16	4	재활용 쇼핑백	생활용품	5000	2023-09-15	1
17	5	빈티지 가죽 백팩	패션	180000	2022-08-01	1
18	6	여름용 린넨 셔츠	패션	45000	2023-04-20	1

19	7	친환경 주방 세트	생활용품	60000	2022-10-10	0
20	8	고성능 그래픽 카드	전자기기	800000	2023-06-01	1
21	9	어린이를 위한 그림책	도서	18000	2022-03-01	1
22	10	천연 에센셜 오일	헬스/뷰티	25000	2023-05-10	1
23	1	휴대용 빔 프로젝터	전자기기	350000	2023-02-01	1
24	2	게이밍 의자	전자기기	200000	2022-07-20	1
25	3	세계사 탐험	도서	22000	2021-02-28	0

- 일부 컬럼은 제외했다.

인덱스가 필요한 이유

우리는 지금까지 데이터를 효과적으로 조회하고 가공하는 다양한 SQL 문법들을 배웠다. 하지만 실제 세상에서는 문법을 아는 것만으로는 부족하다. 우리 쇼핑몰이 대성공을 거두어, 판매하는 상품 수가 25개가 아니라 50만 개, 100만 개로 늘어났다고 상상해 보자.

이제 고객이 우리 쇼핑몰에서 '게이밍 노트북'을 검색한다. 우리 시스템은 데이터베이스에 다음의 쿼리를 실행할 것이다.

```
SELECT * FROM items WHERE item_name = '게이밍 노트북';
```

데이터가 몇 건 없었을 때는 눈 깜짝할 사이에 끝나던 이 쿼리가, 이제는 3초, 5초, 혹은 그 이상이 걸리기 시작한다. 서비스 속도가 느려지고 고객들은 답답함을 느껴 떠나간다. 분명 쿼리는 똑같은데, 왜 이렇게 느려진 걸까? 데이터베이스 안에서 도대체 무슨 일이 벌어지고 있는 걸까?
물론 데이터가 아주 많아야 이렇게 느려진다. 지금의 샘플 데이터는 많지 않기 때문에 금방 출력될 것이다. 여기서는 데이터가 수백만 수천만 건이라고 가정하고 진행하겠다.

느린 검색의 원인: 풀 테이블 스캔 (Full Table Scan)

인덱스가 없는 테이블에서 특정 데이터를 찾는 과정은, 비유하자면 **100만 페이지짜리 거대한 책에서 특정 단어 하나를 찾기 위해, 책의 첫 페이지부터 마지막 페이지까지 한 장 한 장 넘겨보는 것**과 같다.

데이터베이스는 `item_name` 컬럼에 '게이밍 노트북'이라는 값이 어디에 있는지 알 수 있는 아무런 '힌트'가 없다. 그래서 데이터베이스는 가장 무식하고 정직한 방법을 선택한다. 바로 `items` 테이블 전체를 디스크에서 메모리로 읽어 들인 후, 첫 번째 행부터 마지막 100만 번째 행까지 하나씩 차례대로 `item_name` 컬럼의 값을 비교하는 것이다.

[items 테이블 전체 내용, 일부 컬럼 생략]

item_id	seller_id	item_name	category	price	registered_date
1	1	무선 기계식 키보드	전자기기	120000	2022-01-20
2	1	4K UHD 모니터	전자기기	450000	2022-02-15
3	2	프리미엄 게이밍 마우스	전자기기	80000	2021-11-10
4	3	관계형 데이터베이스 입문	도서	30000	2020-05-01
5	4	친환경 세제	생활용품	15000	2023-08-01
6	5	고급 가죽 지갑	패션	70000	2022-06-25
7	1	스마트 워치	전자기기	250000	2023-03-10
8	6	캐시미어 스웨터	패션	95000	2023-10-05
9	7	아로마 디퓨저	생활용품	40000	2022-09-01
10	8	게이밍 노트북	전자기기	1500000	2023-01-30
11	9	SQL 마스터 가이드	도서	35000	2021-04-12
12	10	유기농 비누 세트	헬스/뷰티	20000	2023-02-20
...	...	...	...	...	...

이러한 작업 방식을 **풀 테이블 스캔(Full Table Scan)**이라고 부른다.

- **최선의 경우**: 우리가 찾는 데이터가 우연히 첫 번째 행에 있다면 한 번의 비교만으로 끝난다.
  - 예) 무선 기계식 키보드
- **최악의 경우**: 우리가 찾는 데이터가 맨 마지막 행에 있거나, 아예 존재하지 않는다면 100만 번의 비교를 모두 수

행해야만 결과를 알 수 있다.

풀 테이블 스캔은 빅오 표기법으로 `O(n)`으로 표현한다. 여기서 `n`은 레코드 건수를 나타내며, 레코드 수가 두 배로 증가하면 스캔 시간도 대략 두 배로 증가한다는 의미다. 쉽게 이야기해서 풀 테이블 스캔은 데이터 건 수 만큼 데이터를 탐색해야 한다는 뜻이다.

이렇듯 풀 테이블 스캔에서 데이터가 많아질수록 검색 시간이 정비례해서 늘어나는 것은 당연한 결과다. 이것이 바로 우리의 서비스가 느려진 근본적인 원인이다.

이런 방식을 계속 유지한다면 우리 서비스는 장사가 잘 되면 잘 될수록, 그래서 데이터가 많아 질 수록 점점 더 느려지게 된다. 나중에는 사용자들이 답답해서 우리 서비스를 찾지 않게 될 것이다. 이러한 성능 문제는 반드시 해결해야 한다.

데이터 양에 따른 성능 저하

데이터 양에 따른 정확한 응답시간을 예측하는 것은 거의 불가능하다. 왜냐하면 응답시간은 데이터베이스 서버의 사양(CPU, 메모리, 디스크 I/O 성능), 환경 설정, 데이터의 한 행(Row)당 크기, 시스템의 현재 부하 등 수많은 변수에 따라 크게 달라지기 때문이다.

하지만 **일반적인 기업용 서버 환경(SSD 디스크 기반)을 가정**하고, **한 행당 약 1KB의 데이터를 가진 테이블**을 기준으로 대략적인 예상 응답시간을 정리하면 다음과 같다.

데이터 건수에 따른 풀테이블 스캔 예상 응답시간

데이터 건수	총 데이터 크기 (행당 1KB 가정)	예상 응답시간	비고
100만 건	약 1GB	**수 초 (예: 0.2초 ~ 3초)**	디스크 I/O 속도에 따라 좌우되며, 메모리(버퍼 캐시)에 데이터가 일부라도 있으면 더 빨라짐
1,000만 건	약 10GB	**수 초 ~ 1분 내외 (예: 2초 ~ 30초)**	본격적으로 디스크 I/O 병목이 발생하기 시작하는 구간

1억 건	약 100GB	**수십 초 이상 (예: 20초 ~ 5분)**	다른 작업과 경합이 발생할 경우 응답시간은 예측하기 어려울 정도로 길어질 수 있음

> ⚠ **이 표는 단순 참고용이며, 실제 환경에서는 몇 배 이상 차이가 날 수 있다.**
> - 실무에서는 본인 환경에 맞는 성능 테스트를 통해 성능을 확인해야 한다.
> - 실제로는 수 많은 요청을 동시에 처리한다. 예를 들어서 10명의 사용자가 동시에 요청하면 풀테이블 스캔도 10개가 동시에 발생할 수 있다. 이런 경우 성능은 훨씬 더 느려진다.

실무 이야기1

오늘날의 사용자들은 단 몇 초의 로딩 시간도 기다리지 않는다.

> **페이지 로딩 속도에 관한 연구**
> 페이지 로딩 속도에 관한 다양한 논문들이 있는데 대략 정리하면 다음과 같다.
> - 모바일 페이지 로딩 속도가 1초에서 3초로 늘어나면 이탈률이 32% 증가한다.
> - 모바일 페이지 로딩 속도가 1초에서 5초로 늘어나면 이탈률이 90% 이상 증가한다.
>
> 정리하면 웹 서비스는 최소한 3초 이내, 이상적으로는 1~2초 이내의 빠른 로딩 속도를 목표로 해야 한다.

일반적인 서비스를 생각해보자. 하나의 화면에 단순히 `items` 테이블 하나만 조회할까? 아니다. 보통 한 화면을 보여주기 위해서는 다양한 데이터가 필요하다. 따라서 다양한 테이블들을 함께 조회한다. 이 테이블들이 모두 풀 테이블 스캔이라면 데이터의 양이 적을 때는 문제가 없어 보이겠지만, 데이터가 점점 증가하기 시작하는 순간 서비스도 점점 느려지며 많은 사용자들이 이탈할 것이다.

실무 이야기2

풀테이블 스캔은 매우 비용이 높은 작업이므로, 서비스의 핵심 기능에서 가급적 풀테이블 스캔이 발생하지 않도록 설계하는 것은 필수다.

- **인덱스 활용**: `WHERE` 절에 자주 사용되는 검색 조건 컬럼에는 **인덱스(Index)**를 생성하여 풀테이블 스캔을 방지하는 것이 가장 기본적인 해결책이다.
- **실행 계획 확인**: 쿼리 실행 전 **실행 계획(Execution Plan)**을 확인하여 의도치 않은 풀테이블 스캔이 발생하는지 반드시 점검해야 한다.

- **작업 시간 분리**: 대용량 데이터에 대한 전체 스캔이 불가피한 통계/배치 작업이라면, 서비스 이용자가 적은 새벽 시간에 실행하는 것을 권장한다.

다음 시간부터 인덱스에 대해서 알아보자.

인덱스 소개

그렇다면 이 무식한 풀 테이블 스캔을 피할 방법은 없을까?
다시 책의 비유로 돌아가 보자. 우리는 책에서 특정 단어를 찾을 때 책을 처음부터 다 읽지 않는다. 책의 맨 뒤에 있는 **'찾아보기(색인)'** 페이지를 활용한다.

참고로 찾아보기를 다른 말로 색인이라 한다. 색인은 영어로 인덱스(INDEX)이다.

'찾아보기' 페이지는 어떻게 구성되어 있는가?

1. 책의 중요한 키워드들이 **알파벳순, 가나다순(정렬된 순서)**으로 정리되어 있다.
2. 각 키워드 옆에는 그 키워드가 등장하는 **페이지 번호**가 적혀 있다.

우리는 이 '찾아보기' 덕분에, 찾고 싶은 단어가 어느 페이지에 있는지 아주 빠르게 알아낸 뒤, 해당 페이지로 곧장 점프할 수 있다.
그런데 여기서 중요한 점이 있다. 우리는 찾아보기 페이지를 보고 어떻게 빨리 데이터를 찾을 수 있을까? 바로 중요한 키워드들이 **가나다순(정렬된 순서)**로 정리되어 있기 때문이다!

데이터베이스의 **인덱스(INDEX)**는 이 책의 '찾아보기'와 정확히 동일한 역할을 한다.

인덱스는 특정 컬럼(들)의 데이터를 기반으로 생성되는, 원본 테이블과는 별개의 **특수한 자료 구조**다.

- 인덱스는 지정된 컬럼(예: `item_name`)의 값과, 해당 값을 가진 실제 데이터 행의 위치(예: 주소값, 포인터, PK 값 등)를 한 쌍으로 저장한다.
- 가장 중요한 것은, 인덱스 내부의 데이터는 **항상 정렬된 상태를 유지**한다는 점이다.

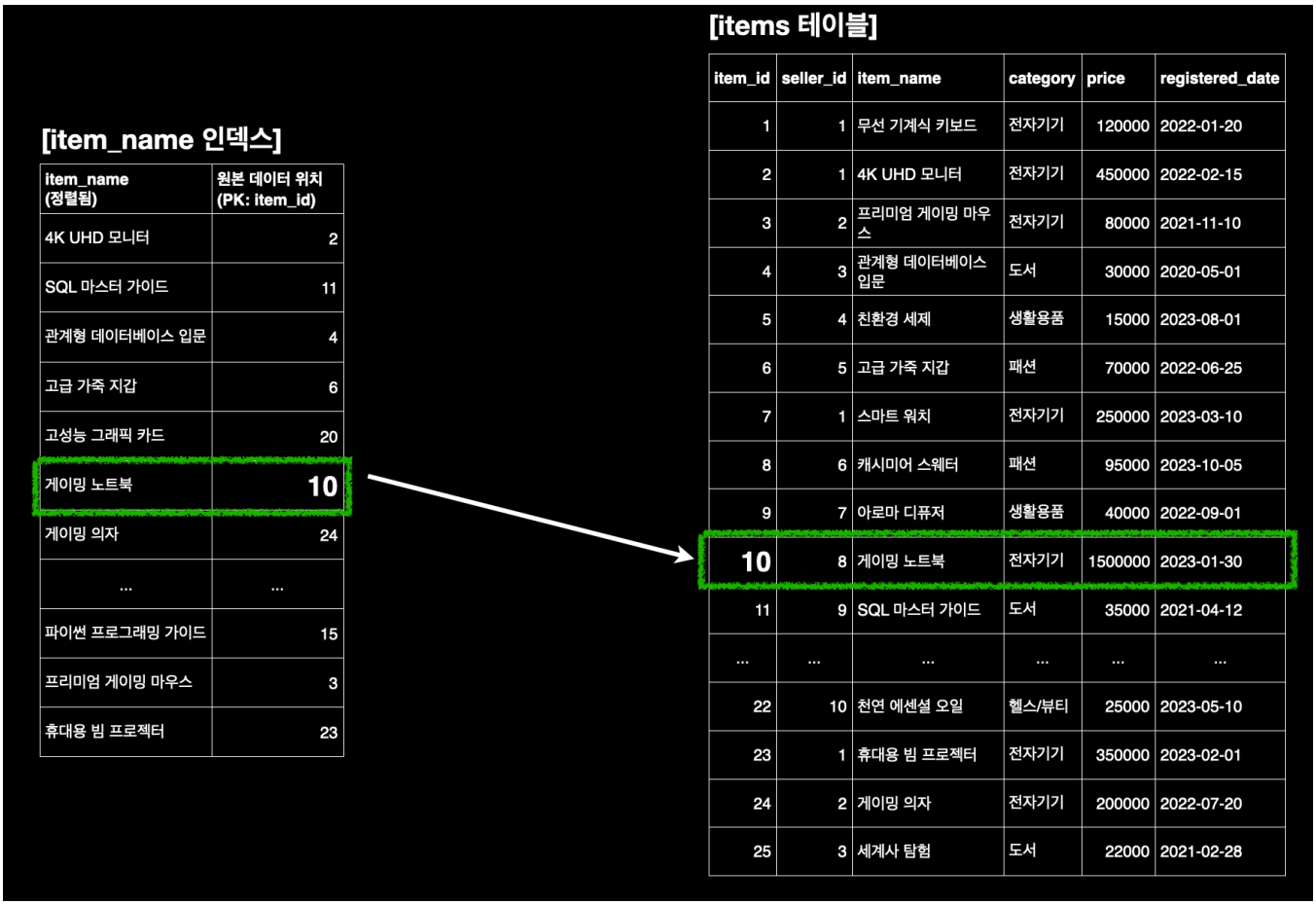
`item_name`을 기반으로 인덱스를 만들면 다음과 같은 구조를 가진다.

[item_name 기반의 인덱스]

item_name (정렬됨)	원본 데이터 위치 (PK: item_id)
4K UHD 모니터	2
SQL 마스터 가이드	11
관계형 데이터베이스 입문	4
고급 가죽 지갑	6
고성능 그래픽 카드	20
게이밍 노트북	10
게이밍 의자	24
노이즈 캔슬링 헤드폰	13
무선 기계식 키보드	1
빈티지 가죽 백팩	17
세계사 탐험	25
스마트 워치	7
아로마 디퓨저	9
어린이를 위한 그림책	21
여름용 린넨 셔츠	18
유기농 비누 세트	12
인체공학 키보드	14
재활용 쇼핑백	16
천연 에센셜 오일	22
친환경 세제	5
친환경 주방 세트	19
캐시미어 스웨터	8
파이썬 프로그래밍 가이드	15

프리미엄 게이밍 마우스	3
휴대용 빔 프로젝터	23

- `item_name`이 가나다라 순으로 정렬되어 있는 것을 확인할 수 있다.
- 정렬된 순서를 보면 숫자, 영문, 가나다라 순으로 정렬된다.
  - 4K UHD 모니터
  - SQL 마스터 가이드
  - 관계형 데이터베이스 입문
  - ...
- 이렇게 정렬된 상태면 사람도 원하는 데이터를 빨리 찾을 수 있듯이 데이터베이스도 원하는 데이터를 빨리 찾을 수 있다.
  - 정렬된 데이터에서 데이터베이스가 원하는 데이터를 어떻게 빨리 찾을 수 있는지 그 원리는 뒤에서 설명한다.
- 원본 데이터의 위치에는 해당 값을 가진 실제 데이터 행의 위치(예: 주소값, 포인터, PK 값 등)를 저장한다. 이 값을 통해 원본 데이터에 빠르게 접근할 수 있다.
  - 예시에서는 보기 쉽게 편의상 PK 값을 적어두었다. 실제로 저장되는 값은 데이터베이스와 인덱스의 종류에 따라서 다르다.



`item_name` 컬럼에 인덱스가 생성된 후, 다시 검색 쿼리를 실행하면 데이터베이스는 완전히 다르게 동작한다.

```
SELECT * FROM items WHERE item_name = '게이밍 노트북';
```

1. `items` 테이블 전체를 순서대로 스캔하는 대신, 먼저 `item_name` 기반의 **인덱스**를 찾아간다.
2. 인덱스는 정렬되어 있으므로, '게이밍 노트북'이라는 값을 아주 빠르게 찾아낸다. (원리는 뒤에서 설명)
3. 찾아낸 인덱스 항목에서 실제 데이터 행의 위치를 확인한다.
4. 그 위치를 이용해 원본 테이블의 해당 위치로 **점프**해서, 단번에 원하는 데이터를 가져온다. (예시에서는 편의상 PK 값을 적어두었지만 해당 행의 실제 위치 값을 가지고 있다고 생각하면 된다.)

인덱스 덕분에 100만 번의 비교 작업이 단 몇 번의 작업으로 줄어들게 된다. 데이터가 1억 건으로 늘어나도 검색 속도는 거의 차이가 나지 않을 정도로 비약적인 성능 향상이 일어난다.

이것이 바로 우리가 인덱스를 사용해야 하는 이유다. 인덱스는 느려진 데이터베이스에 날개를 달아주는, 성능 최적화의 가장 기본적이고 핵심적인 기술이다.

실제 인덱스의 구현

이번 강의에서는 특정 데이터베이스 시스템(예: MySQL)에 종속된 복잡한 구현 세부사항은 다루지 않고, 일반적이고 논리적인 인덱스의 개념을 중심으로 설명한다. 참고로 실제 물리적인 인덱스는 그 종류에 따라 구현이 다 다르다.

예를들어 MySQL의 경우 클러스터 인덱스(Clustered Index)와 보조 인덱스(Secondary Index)라는 2가지 종류의 인덱스를 제공한다.
클러스터 인덱스는 기본 키(PK)를 기반으로 만드는 인덱스이다. 클러스터 인덱스는 원본 데이터 자체를 인덱스에 함께 보관해서 원본 데이터를 매우 빠르게 찾을 수 있다.
보조 인덱스는 원본 데이터의 기본 키(PK) 값을 함께 보관한다. 그리고 이 기본 키(PK) 값으로 클러스터 인덱스를 통해 원하는 데이터를 조회한다.

이처럼 인덱스의 종류에 따라 내부 동작 방식은 다양하지만, 핵심적인 목적은 변함없이 데이터 검색 속도를 향상시키고 쿼리 성능을 최적화하는 것이다. 마치 책의 목차나 찾아보기가 특정 내용을 빠르게 찾을 수 있도록 돕는 것처럼, 데이터베이스 인덱스는 방대한 양의 데이터 속에서 필요한 정보를 효율적으로 찾아낼 수 있도록 돕는 특별한 데이터 구조라고 이해하면 된다.

클러스터 인덱스, 보조 인덱스와 같은 실제 물리적인 인덱스가 어떻게 작동하고, 또 왜 이렇게 나누어 두었는지 궁금할

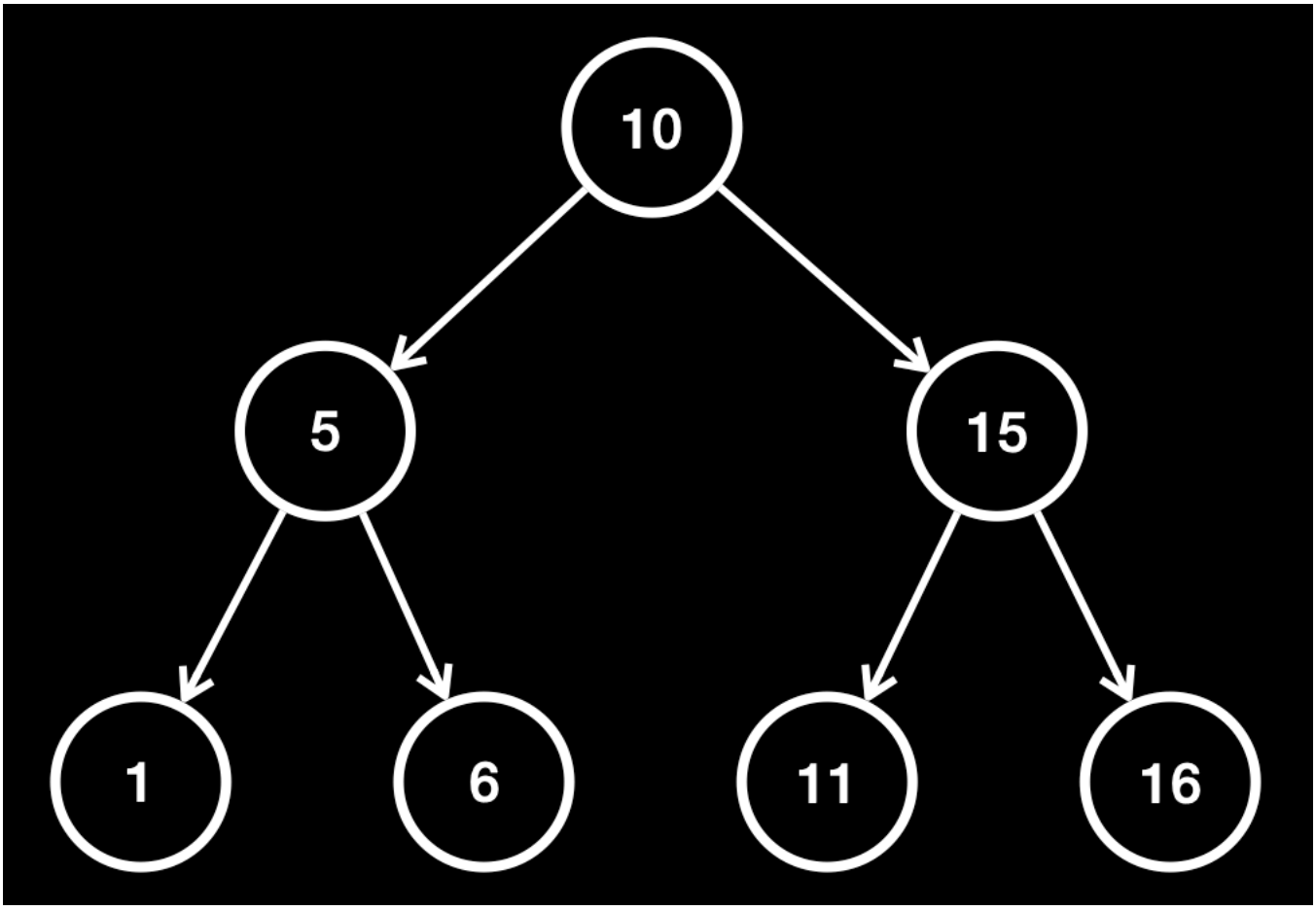
수 있다.
지금 단계에서는 **'특정 컬럼을 정렬해서 저장하고, 이를 통해 원본 데이터에 빠르게 접근할 수 있는 특별한 목차'**라는 인덱스의 큰 그림을 이해하면 충분하다.

> **클러스터 인덱스와 보조 인덱스처럼 인덱스의 각 구현별 차이점과 활용 방안은 실전 데이터베이스 성능 최적화 강의에서 자세히 다루겠다.**

그렇다면 이 마법 같은 인덱스는 내부적으로 어떤 방식으로 만들어져 있기에 이렇게 빠른 검색이 가능한 걸까? 이 원리를 이해하려면 먼저 트리 자료 구조에 대해서 알아야 한다.

트리 자료 구조

인덱스가 어떤 방식으로 데이터를 빠르게 검색하는지 이해하려면 먼저 트리 자료 구조를 알아야 한다.



- 트리는 부모 노드와 자식 노드로 구성된다.

    - 여기서는 각 원이 노드다. 노드 안에는 데이터와 다음 노드들의 위치가 보관된다.
- 가장 높은 조상을 루트(root)라 한다. 이 그림을 뒤집어보면 왜 트리라고 하는지, 처음을 루트라고 하는지 이해가 될 것이다.
- 자식이 2개까지 올 수 있는 트리를 **이진 트리**라 한다.
- 여기에 노드의 왼쪽 자손은 더 작은 값을 가지고, 오른쪽 자손은 더 큰 값을 가지는 것을 **이진 탐색 트리**라 한다.

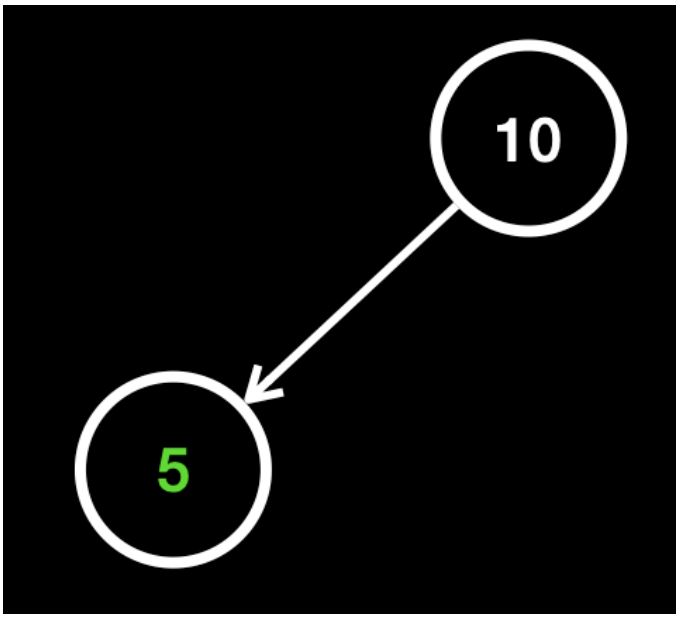
이진 탐색 트리 - 입력 예시

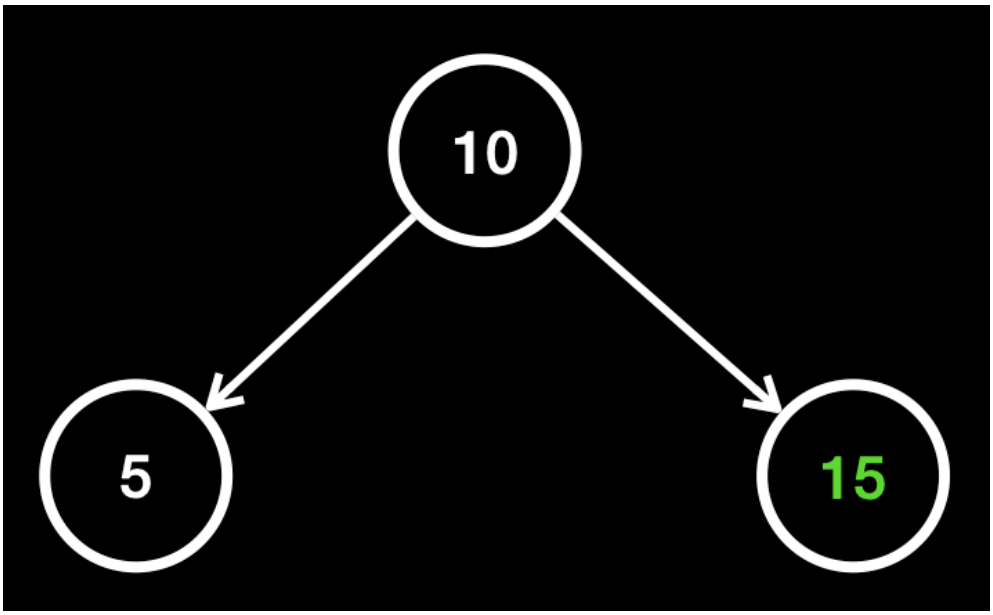
이진 탐색 트리의 핵심은 데이터를 입력하는 시점에 정렬해서 보관한다는 점이다.

그리고 작은 값은 왼쪽에 큰 값은 오른쪽에 저장한다.

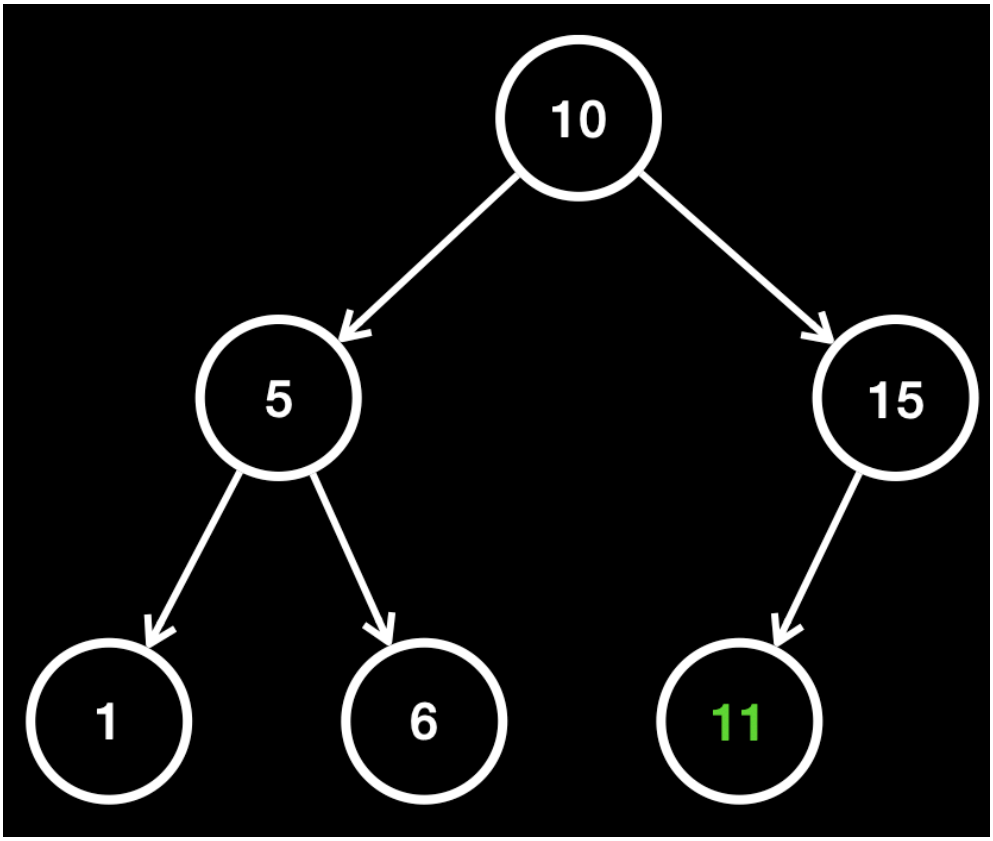
데이터 10, 5, 15, 1, 6, 11, 16을 순서대로 저장한다고 가정해보자.

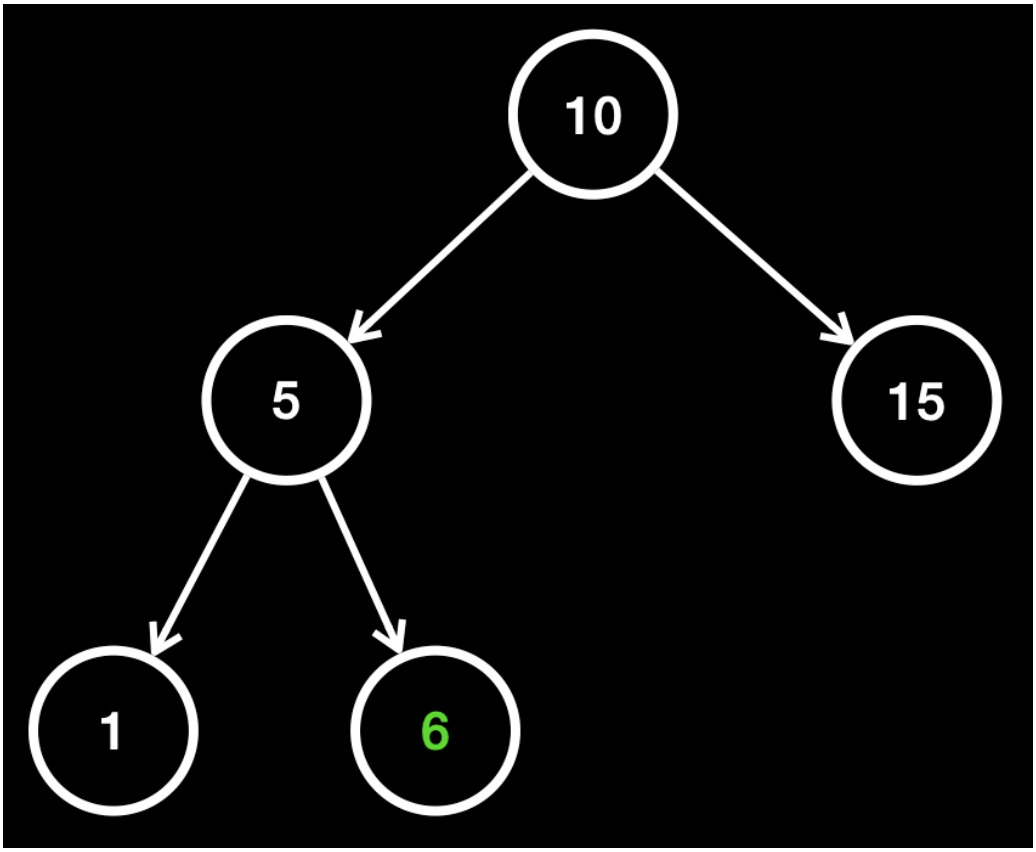
처음에 10을 입력했다고 가정하자. 다음으로 5, 15를 입력한다.



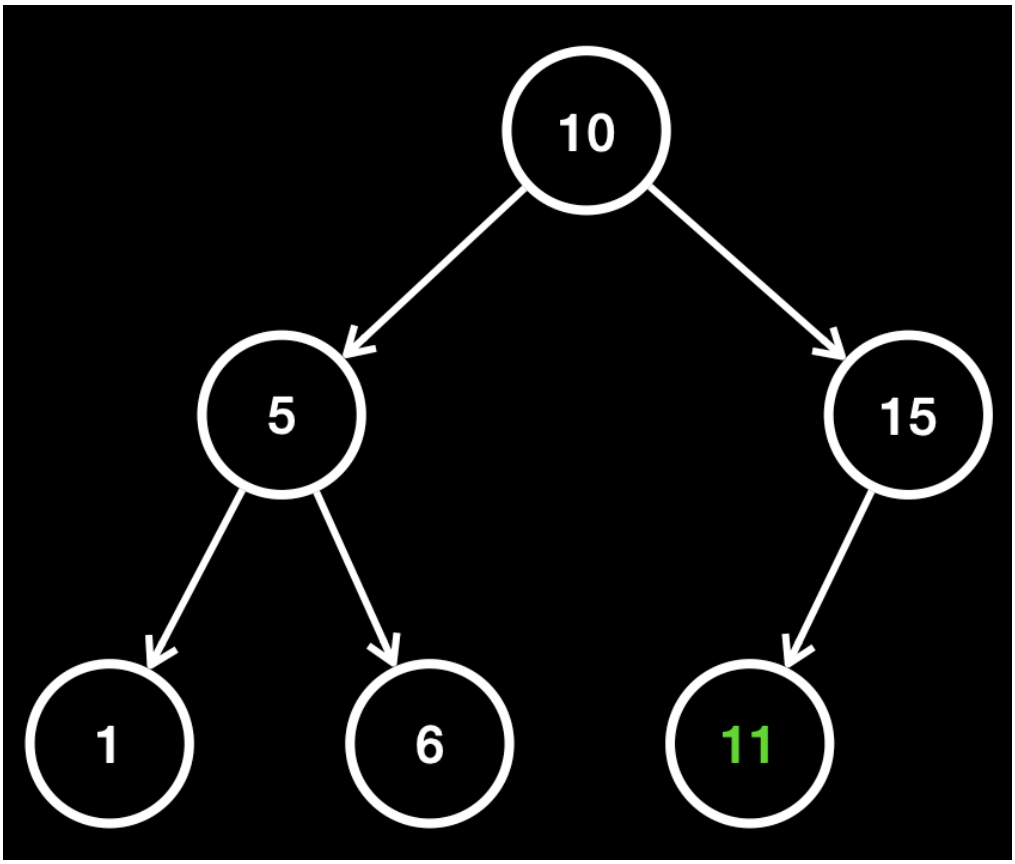


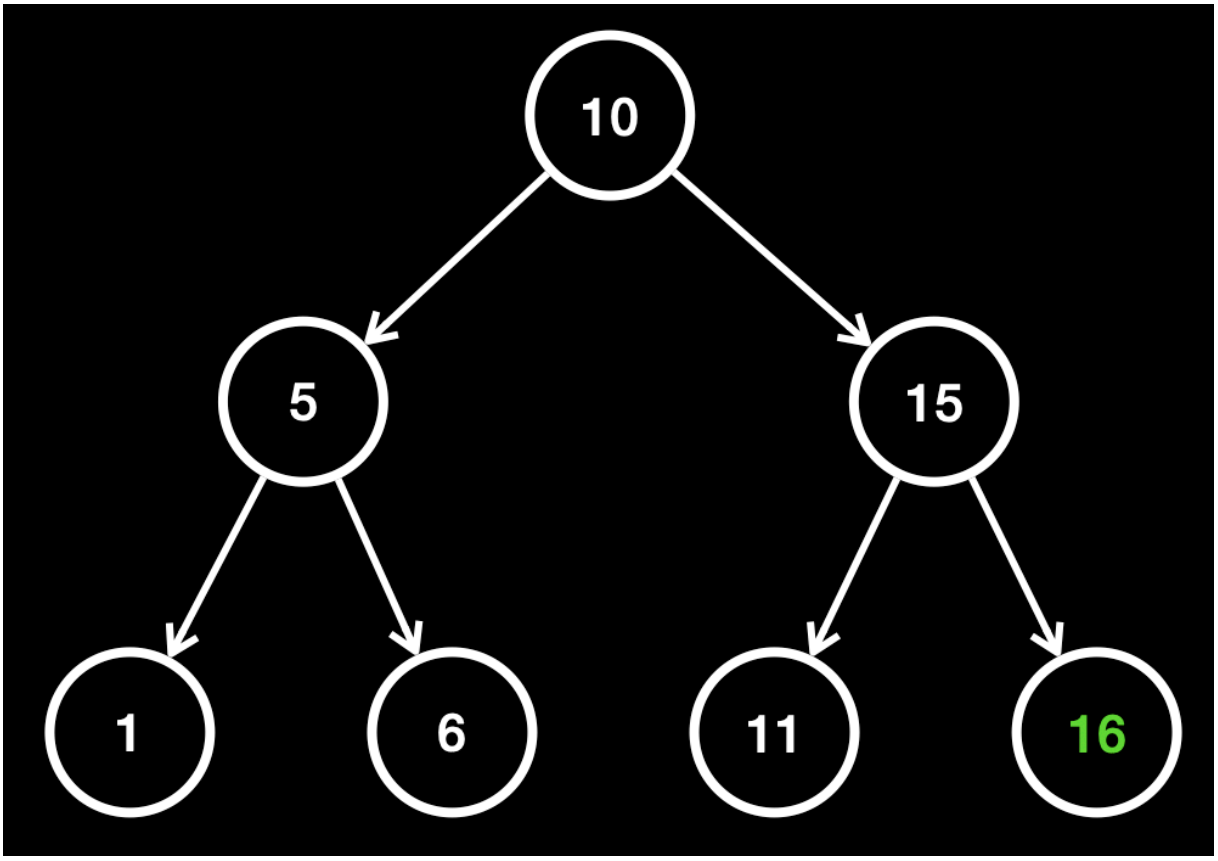
- **5 저장**: 5는 10보다 작으므로 왼쪽에 저장된다.
- **15 저장**: 15는 10보다 크므로 오른쪽에 저장된다.





- **1 저장**: 1은 10보다 작다. 따라서 왼쪽으로 찾아간다. 1은 5보다 작다 따라서 왼쪽에 저장된다.
- **6 저장**: 6은 10보다 작다. 따라서 왼쪽으로 찾아간다. 6은 5보다 크다. 따라서 오른쪽에 저장된다.

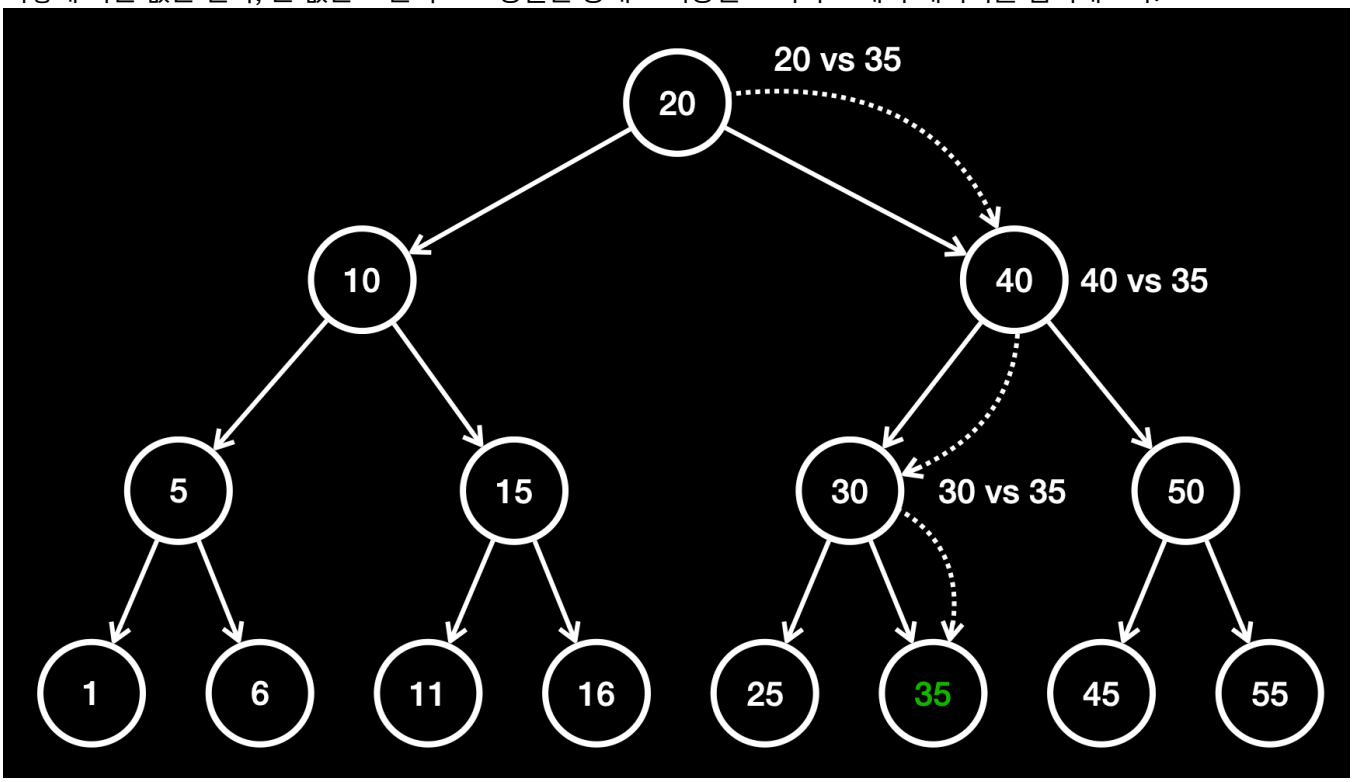




- **11 저장**: 11은 10보다 크다. 따라서 오른쪽으로 찾아간다. 11은 15보다 작다. 따라서 왼쪽에 저장된다.
- **16 저장**: 16은 10보다 크다. 따라서 오른쪽으로 찾아간다. 16은 15보다 크다. 따라서 오른쪽에 저장된다.

이진 탐색 트리 - 검색

이렇게 작은 값은 왼쪽, 큰 값은 오른쪽으로 정렬된 상태로 저장된 트리 구조에서 데이터를 검색해보자.



- 여기에는 총 15개의 데이터가 들어있다. 여기서 숫자 35를 검색한다고 가정해보자.
- 1번: 루트인 20과 35를 비교한다. 35가 더 크므로 오른쪽으로 찾아간다.
- 2번: 40과 35를 비교한다. 35가 더 작으므로 왼쪽으로 찾아간다.
- 3번: 30과 35를 비교한다. 35가 더 크므로 오른쪽으로 찾아간다.
- 4번: 노드에 있는 값을 비교한다. 35와 같으므로 35를 찾는다.

데이터가 총 15개인데 단 4번의 계산만으로 필요한 결과를 얻을 수 있었다.

이진 탐색 트리 계산의 핵심은 한 번에 남은 절반을 날린 다는 점이다. 계산을 단순화 하기 위해 16개의 데이터가 있다고 가정하자.

- 16개의 데이터가 있다. 루트에서 처음 비교를 통해 절반의 데이터를 찾지 않아도 된다. 따라서 16 / 2 = 8이 된다.
- 8개의 데이터가 있다. 비교를 통해 절반의 데이터만 남는다. 따라서 8 /2 = 4가 된다.
- 4개의 데이터가 있다. 비교를 통해 절반의 데이터만 남는다. 따라서 4 / 2 = 2가 된다.
- 2개의 데이터가 있다. 비교를 통해 절반의 데이터만 남는다. 따라서 2 / 2 = 1이 된다.
- 1이 남았으므로 이 값이 맞는지 확인하면 된다.

이진 탐색 트리의 빅오 - O(log n)

16개의 경우 단 4번의 비교 만으로 최종 노드에 도달할 수 있다. 이것을 정리하면 다음과 같다.

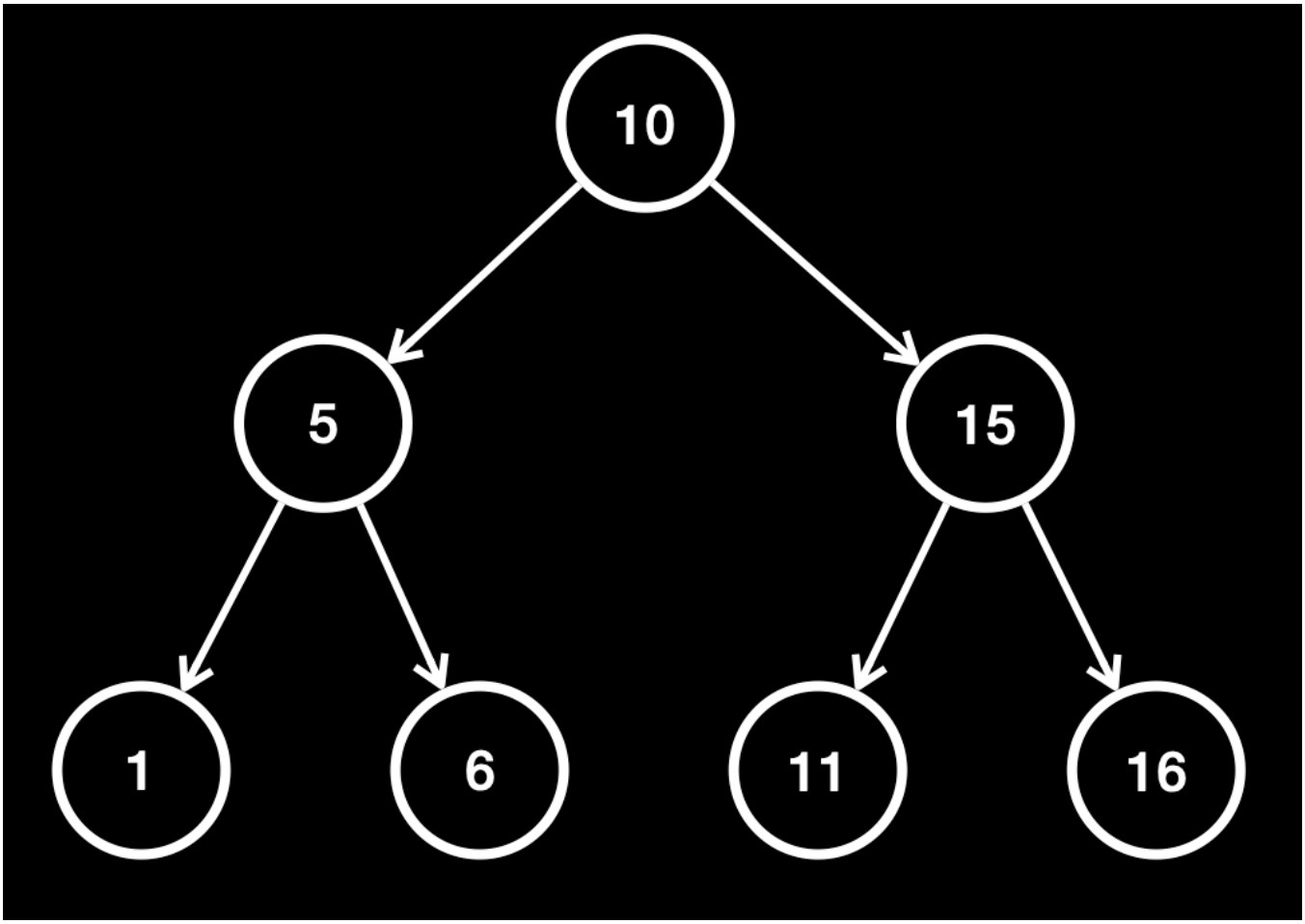
- 2개의 데이터 ➜ 2로 1번 나누기, $\log_2(2)=1$
- 4개의 데이터 ➜ 2로 2번 나누기, $\log_2(4)=2$
- 8개의 데이터 ➜ 2로 3번 나누기, $\log_2(8)=3$
- 16개의 데이터 ➜ 2로 4번 나누기, $\log_2(16)=4$
- 32개의 데이터 ➜ 2로 5번 나누기, $\log_2(32)=5$
- 64개의 데이터 ➜ 2로 6번 나누기, $\log_2(64)=6$
- ...
- 1024개의 데이터 ➜ 2로 10번 나누기, $\log_2(1024)=10$
- 16,384개의 데이터 ➜ 2로 14번 나누기, $\log_2(16384)=14$
- 65,536개의 데이터 ➜ 2로 16번 나누기, $\log_2(65536)=16$
- 131,072개의 데이터 ➜ 2로 17번 나누기, $\log_2(131072)=17$
- 1,000,000개의 데이터 ➜ 2로 약 20번 나누기, $\log_2(1{,}000{,}000)\approx19.93$
- 10,000,000개의 데이터 ➜ 2로 약 24번 나누기, $\log_2(10{,}000{,}000)\approx23.22$
- 100,000,000개의 데이터 ➜ 2로 약 27번 나누기, $\log_2(100{,}000{,}000)\approx26.57$

1024개의 데이터를 단 10번의 계산으로 원하는 결과를 찾을 수 있다. 데이터의 크기가 늘어나도 늘어난 만큼 한 번의 계산에 남은 절반을 날려버리기 때문에 데이터의 크기가 클 수록 효과적이다. 1억개의 데이터가 있어도 약 27번이면 원하는 데이터를 찾을 수 있다!
이 방식을 사용하면 1억개의 데이터가 있어도 처음 비교에서 이미 그 절반인 5000만건이 제거된다!
이게 가능한 이유는 데이터를 정렬한 상태로 보관하기 때문이다. 만약 정렬한 상태가 아니라면 이런 조회 방식은 불가능하다.

이것을 수학으로 $\log_2(n)$ 으로 표현한다. 여기서는 어려운 수학 공식이 핵심이 아니다. 여기서는 쉽게 이야기해서 2로 몇 번 나누어서 1에 도달할 수 있는지를 계산하면 된다.

이진 탐색 트리 - 순회

- 이진 탐색 트리의 핵심은 입력 순서가 아니라, 데이터의 값을 기준으로 **정렬해서 보관**한다는 점이다.
- 정렬해서 보관했기 때문에 정렬된 순서로 데이터를 차례로 조회할 수 있다. (순회 할 수 있다.)
- 데이터를 차례로 순회하려면 중위 순회라는 방법을 사용하면 된다. 왼쪽 서브트리를 방문한 다음, 현재 노드를 처리하고, 마지막으로 오른쪽 서브트리를 방문한다. 이 방식은 이진 탐색 트리의 특성상, 노드를 오름차순(숫자가 점점 커짐)으로 방문한다.



중위 순회 순서

쉽게 이야기해서 자신의 왼쪽의 모든 노드를 처리하고, 자신의 노드를 처리하고, 자신의 오른쪽 모든 노드를 처리하는 방식이다.

- 10의 기준에서 왼쪽 서브트리를 방문한다.
  - 5의 기준에서 왼쪽 서브트리를 방문한다.
    - **1을 출력**한다.
  - **5 자신을 출력**한다.
  - 5의 기준으로 오른쪽 서브트리를 방문한다.
    - **6을 출력**한다.
- **10 자신을 출력**한다.
- 10의 기준에서 오른쪽 서브트리를 방문한다.
  - 15의 기준에서 왼쪽 서브트리를 방문한다.
    - **11을 출력**한다.
  - **15 자신을 출력**한다.
  - 15의 기준으로 오른쪽 서브트리를 방문한다.
    - **16을 출력**한다.

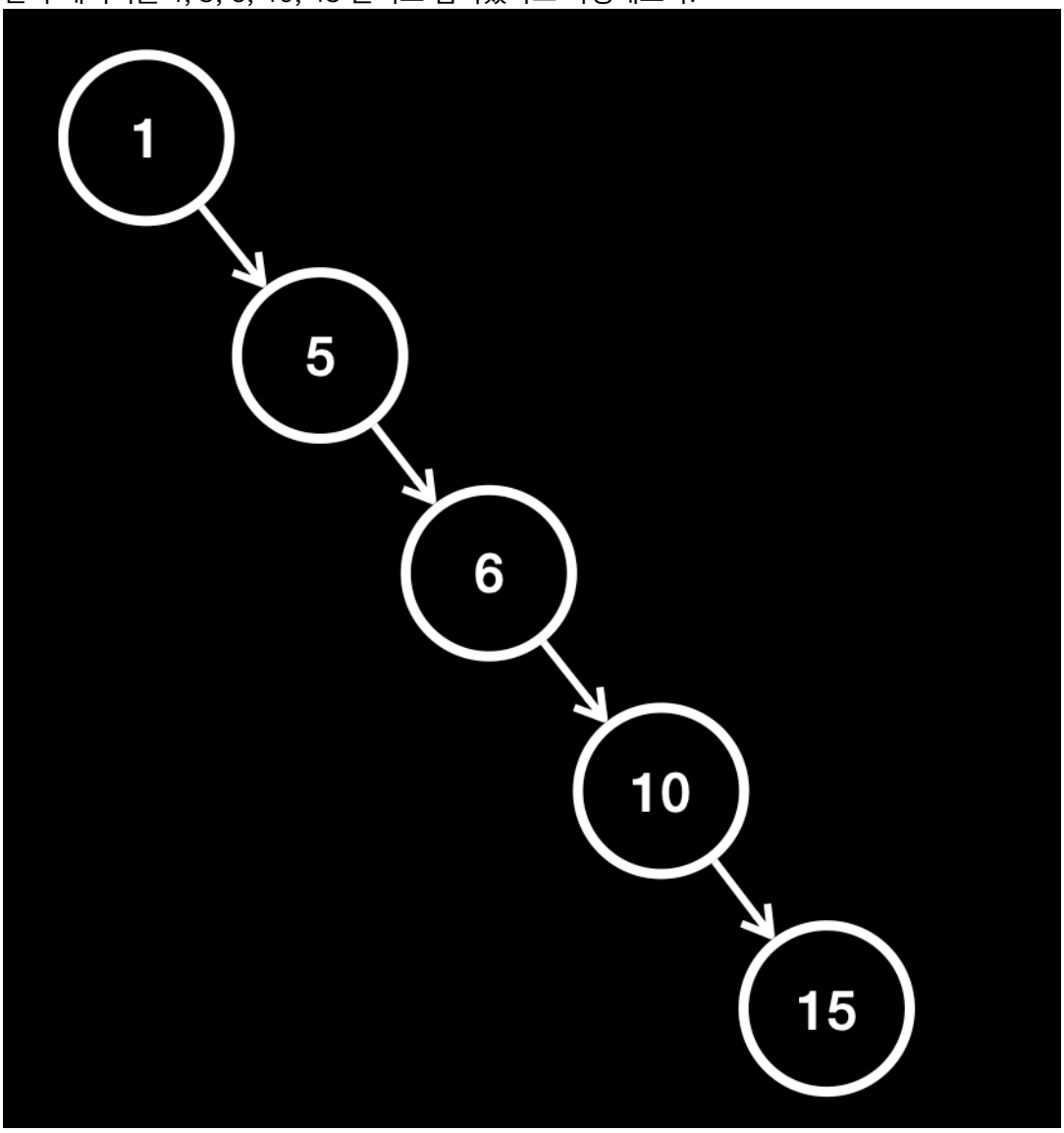
순서대로 오름차순인 1, 5, 6, 10, 11, 15, 16이 출력된 것을 확인할 수 있다.

참고로 자신의 오른쪽 노드부터 순회하면 반대로 내림차순인 16, 15, 11, 10, 6, 5, 1 순으로 출력할 수 있다.

밸런스 트리(Balanced Tree)

이진 탐색 트리의 검색, 삽입, 삭제의 평균 성능은 앞서 살펴본 것 처럼 `O(log n)` 이다. 하지만 트리의 균형이 맞지 않으면 최악의 경우 풀 테이블 스캔과 같은 `O(n)` 의 성능이 나온다. `O(n)` 은 `10` 개의 데이터가 있다면 최악의 경우 `10` 번의 탐색이 필요하다는 뜻이다.

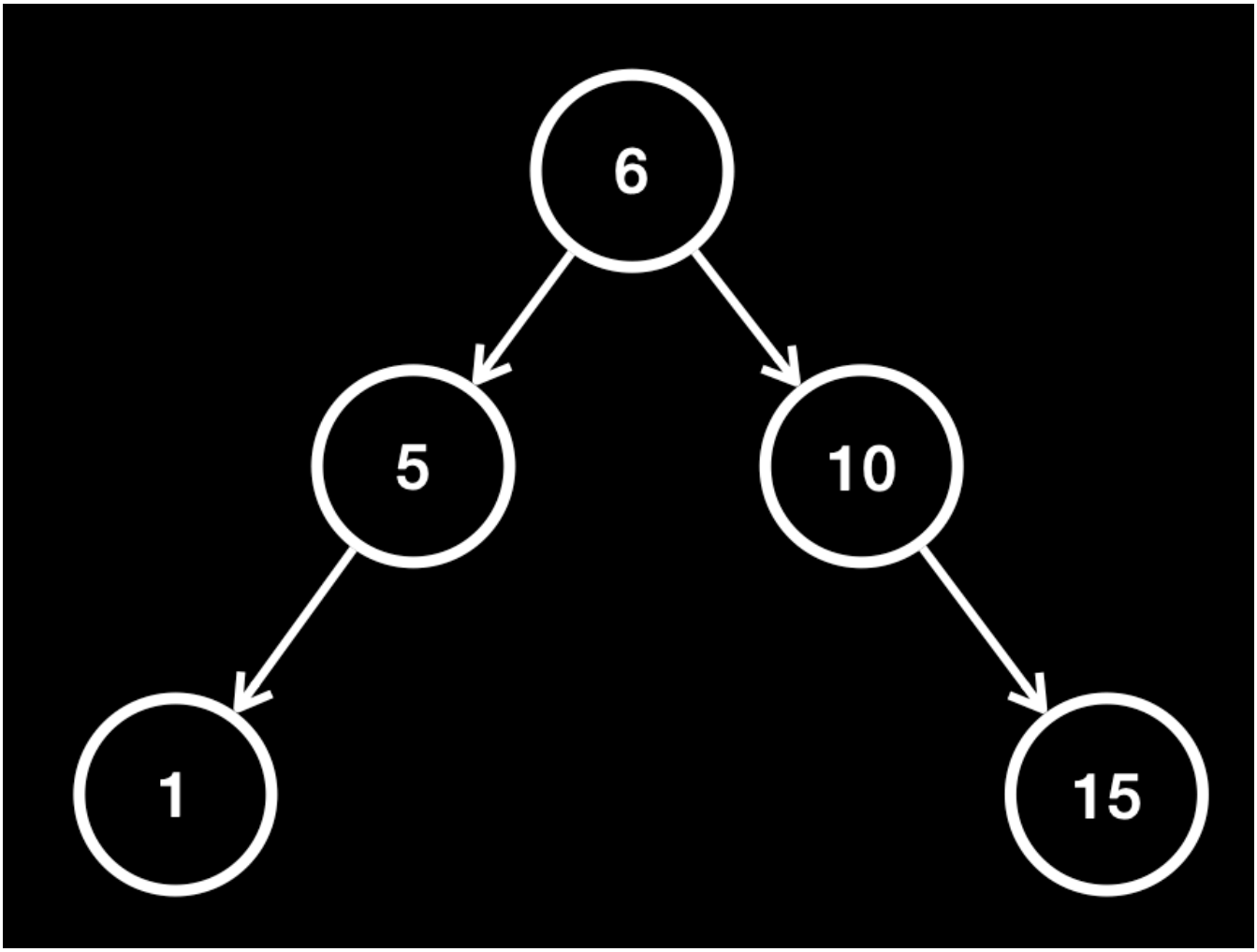
만약 데이터를 1, 5, 6, 10, 15 순서로 입력했다고 가정해보자.



- 이 경우 추가하는 값이 계속 오른쪽으로 입력된다.
- 이렇게 오른쪽으로 치우치게 되면, 결과적으로 15를 검색 했을 때 데이터의 수인 5만큼 검색을 해야 한다.
- 이것은 성능상 풀 테이블 스캔과 같다! 이처럼 최악의 경우 `O(n)` 의 성능이 나온다.

이진 탐색 트리 개선

이런 문제를 해결하기 위한 다양한 해결 방안이 있는데 트리의 균형이 너무 깨진 경우 동적으로 균형을 다시 맞추는 것이다.



- 예를 들어 앞서 본 트리에서 중간에 있는 6을 기준으로 다시 정렬한다.
- 이렇게 균형을 유지하는 트리를 밸런스 트리(Balanced Tree)라고 한다.
- 대부분의 관계형 데이터베이스는 인덱스에 밸런스 트리를 사용해서 균형을 유지한다. 따라서 최악의 경우에도 `O(log n)`의 성능을 제공한다.

실제 인덱스에 사용되는 트리

인덱스는 트리 구조를 사용하기 때문에 정렬된 상태로 저장된다.

예를 들어 `item_name` 컬럼을 기준으로 인덱스를 만들면 다음과 같이 `item_name` 컬럼이 정렬된 상태로 저장된다.

그리고 인덱스에는 인덱스에 사용할 값(`item_name`)과 원본 데이터의 위치가 하나의 쌍으로 저장된다.

이전에 보았던 트리 구조에서 `item_name`과 원본 데이터의 위치를 함께 보관한다고 생각면 된다.

(물론 트리 구조에 데이터를 보관하기 위해 비교할 때는 `item_name` 컬럼만 사용한다.)

[item_name 기준 인덱스]

item_name (정렬됨)	원본 데이터 위치 (PK: item_id)
4K UHD 모니터	2
SQL 마스터 가이드	11
관계형 데이터베이스 입문	4
고급 가죽 지갑	6
고성능 그래픽 카드	20
게이밍 노트북	10
게이밍 의자	24
빈티지 가죽 백팩	17
세계사 탐험	25
...	...

인덱스는 실제로는 B-Tree 구조이지만, 개념적으로는 표와 같이 정렬된 목록으로 이해할 수 있다.

대부분의 주요 관계형 데이터베이스 시스템은 인덱스를 구현하기 위해 앞서 설명한 이진 트리를 개선한 B-트리 또는 B-트리의 변형(B+트리 등)을 사용한다.

이진 탐색 트리는 검색에 `O(log n)`의 성능을 제공하지만, 데이터가 순차적으로 입력되면 한쪽으로 치우쳐지는 문제(Worst Case `O(n)`)가 발생할 수 있다. 이 문제를 해결하기 위해 밸런스 트리(Balanced Tree) 개념이 도입되었고, B-트리는 이러한 밸런스 트리의 한 종류이다.

하지만 B-트리가 특별히 중요한 이유는 단순히 균형을 유지하는 것을 넘어, 데이터베이스처럼 대용량 데이터를 다루는 환경에 최적화되어 있기 때문이다. 데이터베이스는 데이터를 메모리가 아닌 디스크에 저장한다. 디스크에서 데이터를 읽는 작업(Disk I/O)은 메모리에서 읽는 것보다 훨씬 느리다. 따라서 디스크 I/O 횟수를 줄이는 것이 성능에 결정적인 영향을 미친다.

이진 탐색 트리는 한 노드가 하나의 데이터만 가지고 있어, 데이터를 검색할 때 여러 노드를 방문해야 한다면 그만큼 디스크에서 많은 블록을 읽어야 한다. 반면 B-트리는 하나의 노드가 여러 개의 자식 노드를 가질 수 있고, 더 많은 데이터를 저장할 수 있도록 설계되었다. 이는 디스크에서 한 번 데이터를 읽을 때 더 많은 정보를 가져올 수 있게 하여 디스크 I/O 횟수를 획기적으로 줄여준다. 덕분에 B-트리는 디스크 I/O를 최소화하면서 대용량 데이터에서 효율적인 검색 성

능을 제공한다.

> **B-트리와 인덱스 구현에 대한 상세한 내용은 데이터베이스 성능 최적화에서 자세히 다룬다. 지금은 정렬된 인덱스를 사용해서 데이터를 빠르게 찾을 수 있다는 점만 이해하면 충분하다.**

인덱스 생성, 조회, 삭제

이번 시간에는 `CREATE`, `SHOW`, `DROP`이라는 간단한 명령어를 통해 인덱스를 직접 다루는 실습을 진행하자.

`CREATE INDEX`: 인덱스 생성하기

가장 기본이 되는 인덱스 생성 명령어는 `CREATE INDEX`이다. 문법은 다음과 같다.

```
CREATE INDEX 인덱스이름 ON 테이블이름 (컬럼1, 컬럼2, ...);
```

- **인덱스 이름**: 생성할 인덱스에 고유한 이름을 붙여준다. 보통 `idx_테이블명_컬럼명` 과 같은 규칙으로 지으면 관리하기 편하다.
- **테이블 이름**: 인덱스를 생성할 대상 테이블.
- **컬럼**: 인덱스를 구성할 컬럼. 하나 이상의 컬럼을 지정할 수 있다.

이제 실습을 해보자. 우리는 `items` 테이블에서 특정 `item_name`으로 상품을 검색하는 경우가 많다고 가정하고, `item_name` 컬럼에 대한 인덱스를 생성해 보겠다.

```
CREATE INDEX idx_items_item_name ON items (item_name);
```

이 쿼리를 실행하면, 데이터베이스는 `items` 테이블의 모든 `item_name` 값을 읽어 정렬한 뒤, `idx_items_item_name` 인덱스를 디스크에 생성한다. 테이블의 데이터가 많을수록 이 작업은 시간이 오래 걸릴 수 있다.

`SHOW INDEX` : 테이블의 인덱스 정보 확인하기

"테이블에 어떤 인덱스들이 생성되어 있는지 확인하려면 어떻게 해야 할까?"

`SHOW INDEX` 명령어를 사용하면 테이블에 걸려있는 모든 인덱스의 정보를 한눈에 볼 수 있다.

items 인덱스 확인

```
SHOW INDEX FROM items;
```

[실행 결과]

Table	Non_unique	Key_name	Column_name	Cardinality
items	0	PRIMARY	item_id	25
items	1	fk_items_sellers	seller_id	10
items	1	idx_items_item_name	item_name	25

결과를 자세히 살펴보자.

- **Key_name**: 인덱스의 이름이다. 우리가 방금 만든 `idx_items_item_name`이 보인다.
- **Column_name**: 해당 인덱스가 어떤 컬럼을 기반으로 만들어졌는지 보여준다.
- `PRIMARY` **와** `seller_id`: 여기서 흥미로운 점은, 우리는 `item_name` 컬럼에 대한 인덱스만 만들었는데 `PRIMARY`와 `fk_items_sellers`라는 인덱스가 이미 존재한다는 것이다.
  - MySQL에서는 `PRIMARY KEY`(기본 키)나 `FOREIGN KEY`(외래 키) 제약조건을 설정하면, 해당 컬럼에 대해 **자동으로 인덱스를 생성한다.** `item_id`는 기본 키이므로 `PRIMARY` 인덱스가, `seller_id`는 외래 키이므로 `fk_items_sellers` 인덱스가 이미 존재했던 것이다. 매우 중요한 사실이니 꼭 기억해 두자! (`UNIQUE` 제약조건도 마찬가지로 인덱스가 생성된다.)
- **Non_unique**: `1`이면 중복 값을 허용하는 인덱스, `0`이면 중복을 허용하지 않는 고유 인덱스(`UNIQUE` 또는 `PRIMARY KEY`)라는 의미다. `idx_items_item_name`은 `1`이므로, 상품명이 같은 다른 상품이 등록될 수 있다는 것을 알 수 있다.
- **Cardinality**: 인덱스에 저장된 유니크한 값의 개수에 대한 추정치다. 이 값이 높을수록 중복도가 낮다는 의미이며, 인덱스의 성능이 좋다고 판단할 수 있다. (Cardinality는 뒤에서 설명한다.)

sellers 인덱스 확인

```
SHOW INDEX FROM sellers;
```

[실행 결과]

Table	Non_unique	Key_name	Seq_in_index	Column_name	Cardinality
sellers	0	PRIMARY	1	seller_id	10
sellers	0	seller_name	1	seller_name	10

- `PRIMARY` : `seller_id` 기본 키로 인덱스가 자동 생성되었다.
- `seller_name` : `seller_name` 에 UNIQUE 제약조건으로 인덱스가 자동 생성되었다.

유니크 제약조건에 인덱스를 자동 생성하는 이유

유니크 제약조건은 컬럼 내 데이터의 유일성을 보장해야 한다. 따라서 새로운 데이터를 삽입(INSERT)하거나 기존 데이터를 수정(UPDATE)할 때마다, 입력하려는 값이 테이블에 이미 존재하는지 빠르게 확인해야 한다. 만약 인덱스가 없다면 이 중복 검사를 위해 매번 풀 테이블 스캔이 발생하여 쓰기 성능이 크게 저하될 것이다.

> **인덱스 자동 생성**
> MySQL은 `PRIMARY KEY` (기본 키), `FOREIGN KEY` (외래 키), `UNIQUE` 제약조건에 대해서 인덱스를 자동 생성한다.

`DROP INDEX` : 인덱스 삭제하기

시간이 지나 더 이상 사용하지 않거나, 오히려 쓰기 성능에 방해가 되는 인덱스는 삭제해야 한다. `DROP INDEX` 명령어로 간단하게 삭제할 수 있다.

```
DROP INDEX 인덱스이름 ON 테이블이름;
```

방금 만들었던 `idx_items_item_name` 인덱스를 삭제해 보자.

```
DROP INDEX idx_items_item_name ON items;
```

이 쿼리를 실행하면 `idx_items_item_name` 인덱스를 구성하던 데이터 구조가 디스크에서 완전히 사라진다. 이제

`item_name`을 조건으로 검색하면 다시 풀 테이블 스캔이 발생할 것이다. 물론, 원본 `items` 테이블의 데이터에는 아무런 영향이 없다.

인덱스가 정말 사용되는지 확인하는 법 (`EXPLAIN`)

데이터베이스에는 쿼리를 어떤 방식으로 최적화해서 실행할지 계획하는 기능이 있는데, 이것을 쿼리 옵티마이저(최적화기)라 한다.
인덱스를 만들었다고 해서, 데이터베이스가 모든 `SELECT` 문에 항상 그 인덱스를 사용하는 것은 아니다. 데이터의 분포나 쿼리의 형태에 따라, 데이터베이스 옵티마이저는 인덱스를 사용하는 것보다 풀 테이블 스캔이 더 빠르겠다고 판단할 수도 있다. 추가로 사용할 수 있는 인덱스가 여러개 있다면 어떤 인덱스를 사용할지도 선택한다.

우리가 만든 인덱스가 실제로 쿼리에 사용되는지 확인하려면 `EXPLAIN` 이라는 명령어를 쿼리문 앞에 붙여보면 된다.

인덱스가 없을 때 (또는 인덱스를 삭제한 뒤)

```
-- idx_items_item_name 인덱스가 없을 때 실행 결과 확인
EXPLAIN SELECT * FROM items WHERE item_name = '게이밍 노트북';
```

[실행 결과]

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

실행 결과를 통해 데이터베이스가 우리의 쿼리를 어떻게 실행할지 계획했는지, 즉 '실행 계획'을 엿볼 수 있다. 각 항목이 무엇을 의미하는지 자세히 살펴보자.

- `type: ALL` : **가장 중요하게 봐야 할 부분이다.** `type`은 데이터베이스가 테이블에 어떻게 접근할지를 나타낸다. `ALL`은 **풀 테이블 스캔(Full Table Scan)**을 의미한다. 즉, `items` 테이블의 처음부터 끝까지 **모든 데이터를 하나씩 다 읽어서** 조건에 맞는 데이터를 찾는다는 뜻이다. 지금은 데이터가 25개뿐이라 문제가 없지만, 실무에서처럼 데이터가 수백만 건에 달한다면 상상만 해도 끔찍한 성능 저하를 일으키는 주범이 된다.
    - `ALL`이라고 표시되면 풀 테이블 스캔을 의미한다. 즉, 테이블의 모든 행을 처음부터 끝까지 다 읽었다는 뜻이다.
    - 인덱스를 제대로 사용했다면 `ref`, `range` 등 다른 값이 표시된다.
        - `ref`는 `=` 조건이나 `JOIN`에서 인덱스를 사용했다는 의미다.
        - `range`는 범위 검색(`BETWEEN`, `>`, `<`, `>=` 등)에서 인덱스를 사용했다는 의미다.

- `key: NULL`: `key`는 쿼리를 실행할 때 사용한 인덱스를 보여준다. 이 값이 `NULL`이라는 것은 **어떤 인덱스도 사용하지 못했다**는 것을 명확하게 알려준다. `item_name` 컬럼으로 데이터를 찾고 있지만, 해당 컬럼에 인덱스가 없기 때문에 당연한 결과다.

- `rows: 25`: 옵티마이저가 쿼리를 처리하기 위해 **탐색할 것으로 예측하는 행의 수**다. 현재 `items` 테이블의 전체 데이터가 25개이므로, 결국 테이블 전체를 다 훑어보겠다고 말하는 것과 같다. `type`이 `ALL`이니 당연히 전체 행의 개수가 표시된다. 이 값이 작을 수록 효율적인 쿼리라 할 수 있다. (실제 실행하는게 아니라 데이터베이스 나름의 통계 데이터를 기반으로 하는 예측 정보다. 따라서 정확하게 맞는 것은 아니다.)

- `filtered: 10.00`: 테이블에서 읽어온 행들 중에서 `WHERE` 조건으로 필터링되고 난 후, 최종적으로 남을 것으로 예측되는 행의 비율이다. 여기서는 25개의 행을 모두 읽은 후, 그중 10%인 2.5개 정도의 행이 `item_name = '게이밍 노트북'` 조건을 만족할 것이라고 예측하고 있다. (실제 실행하는게 아니라 데이터베이스 나름의 통계 데이터를 기반으로 하는 예측 정보다. 따라서 정확하게 맞는 것은 아니다.)

- `Extra: Using where`: 데이터를 가져온 후에 `WHERE` 절의 조건(`item_name = '게이밍 노트북'`)을 사용해 필터링 작업을 수행했다는 의미다. 만약 인덱스를 효율적으로 사용했다면, 처음부터 조건에 맞는 데이터만 골라서 가져왔을 것이다. 하지만 인덱스가 없으니 일단 모든 데이터를 다 가져와서, 그 후에 조건에 맞는지 일일이 비교하는 비효율적인 방식으로 일하고 있음을 보여준다.

결론적으로, 이 실행 계획은 `item_name` 컬럼에 인덱스가 없어서 `items` 테이블의 모든 행(25개)을 전부 스캔하는, 매우 비효율적인 방식으로 쿼리가 동작할 것임을 알려준다. 이제 왜 우리가 인덱스를 만들고, 그것이 제대로 사용되는지 확인해야 하는지 확실히 이해했을 것이다.

> `EXPLAIN` 각 항목에 대한 더 자세한 내용은 데이터베이스 성능 최적화 강의에서 다룬다.

인덱스와 동등 비교

데이터베이스에서 인덱스는 다음 세 가지 상황에 사용된다.

- 동등 비교(`=`)
- 범위 검색(`BETWEEN`, `>`, `<`, `>=`, `LIKE` 등)
- `ORDER BY`를 통한 정렬 작업

인덱스와 동등 비교 시작

`type: ref`는 동등 비교(`=`) 조건이나 `JOIN`에서 인덱스를 사용했다는 의미다.

`EXPLAIN`을 사용해서 동등 비교를 확인해보고, 인덱스가 있을 때와 없을 때의 차이를 살펴보자.

인덱스가 있을 때

먼저 `items` 테이블의 `item_name` 컬럼에 인덱스를 생성한다.

```
CREATE INDEX idx_items_item_name ON items (item_name);
```

이제 `EXPLAIN`으로 쿼리 실행 계획을 확인해 보자.

```
EXPLAIN SELECT * FROM items WHERE item_name = '게이밍 노트북';
```

- `idx_items_item_name` 인덱스가 사용되었다. 그림에서는 줄여서 `item_name` 인덱스라고 표현했다.

> [영상 정정]

> 영상에서는 이 그림의 `item_name` 인덱스의 한글 정렬이 잘못되어 있다. 메뉴얼의 그림에 수정해두었다.

[실행 결과]

id	type	possible_keys	key	rows	filtered	Extra
1	ref	idx_items_item_name	idx_items_item_name	1	100.00	NULL

인덱스를 생성한 후의 실행 계획은 이전과 확연히 달라진 것을 볼 수 있다. 하나씩 분석해 보자.

- `type: ref` : 이전의 `ALL` 과 비교했을 때 가장 극적인 변화다. `type` 이 `ref` 라는 것은, **인덱스를 사용해 동등 비교(`=`) 조건으로 데이터를 찾았다**는 의미다. `ref` 는 'reference(참조)'의 약자로, 인덱스를 통해 조건에 맞는 데이터를 매우 효율적으로 참조해서 가져왔다는 뜻이다. 풀 테이블 스캔(`ALL`)과는 비교할 수 없을 정도로 빠른 접근 방식이다.

- `possible_keys` : 현재 쿼리에서 사용 가능한 인덱스의 후보이다. 지금은 `idx_items_item_name` 하나만 있지만 현재 쿼리에서 사용가능한 인덱스를 모두 보여준다. 이 후보들 중에 선택되어 사용될 인덱스가 다음 `key` 항목에 나타난다.

- `key: idx_items_item_name` : 이전에는 `NULL` 이었던 이 값에 우리가 방금 생성한 인덱스의 이름(`idx_items_item_name`)이 명확하게 표시된다. 이것은 옵티마이저가 이 쿼리를 실행하는 데 `idx_items_item_name` **인덱스를 사용했음**을 보여주는 직접적인 증거다.

- `filtered: 100.00` : 인덱스를 통해서 찾은 1개의 행을 100% 선택한다는 뜻이다.

- `rows: 1` : 이 또한 엄청난 변화다. 인덱스가 없을 때는 테이블 전체 행의 수인 `25` 를 스캔할 것으로 예측했지만, 이제는 단 `1` **개의 행만 읽으면 된다**고 예측한다. 마치 책의 맨 뒤에 있는 찾아보기를 통해 '게이밍 노트북'이라는 단어가 있는 페이지를 바로 찾아가는 것과 같다. 테이블 전체를 뒤지는 것이 아니라, 인덱스를 통해 필요한 데이터의 위치를 정확히 찾아가기 때문에 탐색하는 행의 수가 극적으로 줄어든다.

> `EXPLAIN` **은 실제 SQL 쿼리를 실행하는 것이 아니다. 여기서** `rows` **는 예측값이다. 따라서 환경에 따라 1이 아닌 다른 값이 나올 수 도 있다.**

- `Extra: NULL` : 이전에 표시되었던 `Using where` 가 사라졌다. 이는 **인덱스 단계에서 이미 모든 검색 조건이**

충족되었기 때문에, 데이터를 가져온 후 별도의 필터링 작업이 필요 없었다는 것을 의미한다. 그만큼 작업이 더 단순하고 효율적으로 처리된 것이다.

결론적으로, `item_name` 컬럼에 인덱스를 생성하자 데이터베이스 옵티마이저는 풀 테이블 스캔이라는 비효율적인 방법을 버리고, 인덱스를 사용해 단 하나의 행만 읽어오는 매우 효율적인 실행 계획을 세웠다. 이것이 바로 우리가 인덱스를 사용하는 핵심적인 이유다.

> 중요
> 테이블에 데이터가 몇 만 건 이상이면 인덱스를 사용하는게 이득이지만, 지금처럼 샘플 데이터가 너무 적은 경우 데이터베이스는 환경에 따라 인덱스를 사용하지 않고, 그냥 풀 테이블 스캔을 선택할 수도 있다. 예를 들어 2페이지 정도의 작은 책이라면 색인을 찾기 보다 그냥 책을 바로 보는게 원하는 결과를 더 빨리 얻을 수도 있기 때문이다.
>
> 이럴 때 인덱스를 강제로 적용하려면 다음과 같이 FORCE INDEX를 사용하면 된다. 그러면 인덱스를 사용한 실행 계획을 확인할 수 있다.
>
> ```
> EXPLAIN SELECT * FROM items FORCE INDEX (idx_items_item_name)
> WHERE item_name = '게이밍 노트북';
> ```
>
> 이 방법을 사용하면 쿼리 옵티마이저가 최적의 인덱스를 선택할 수 없기 때문에 실무에서는 권장하지 않는다. 꼭 필요하다면 주의해서 사용해야 한다.

인덱스와 범위 검색

앞에서 `type`이 `range`일 때 범위 검색(`BETWEEN`, `>`, `<`, `LIKE` 등)에 인덱스가 사용된다고 설명했다. 이번에는 `items` **테이블의** `price` **컬럼**을 사용하여 범위 검색(`BETWEEN`)에 인덱스가 어떻게 사용되는지 `EXPLAIN`으로 확인해 보자.

인덱스가 없을 때

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

[실행 결과]

id	type	key	rows	filtered	Extra
1	ALL	NULL	25	11.11	Using where

`price` 컬럼에 인덱스가 없는 상태에서 범위 검색을 실행한 결과, 비효율적인 실행 계획이 세워졌다.

- `type: ALL` : **풀 테이블 스캔**이 발생했다. 데이터베이스는 `price`가 `50000`에서 `100000` 사이인 상품을 찾기 위해 `items` 테이블의 모든 상품 데이터를 처음부터 끝까지 하나씩 확인해야만 한다.
- `key: NULL` : `price` 컬럼을 조건으로 사용했지만, 이 컬럼에는 인덱스가 없으므로 사용된 인덱스가 없다는 의미의 `NULL`이 표시된다.
- `rows: 25` : 테이블의 전체 행 수인 `25`가 표시된다. 풀 테이블 스캔을 하므로 당연한 결과다. (예측치 이므로 달라질 수 있다.)
- `filtered: 11.11` : 스캔한 25개의 행 중에서 `WHERE price BETWEEN 50000 AND 100000` 조건을 만족하는 행은 약 11.11%일 것이라고 옵티마이저가 예측하고 있다. (예측치 이므로 달라질 수 있다.)
- `Extra: Using where` : 데이터를 가져온 후에 `WHERE` 절의 조건을 사용해 필터링 작업을 수행했다는 의미다. 만약 인덱스를 효율적으로 사용했다면, 처음부터 조건에 맞는 데이터만 골라서 가져왔을 것이다. 하지만 인덱스가 없으니 일단 모든 데이터를 다 가져와서, 그 후에 조건에 맞는지 일일이 비교하는 비효율적인 방식으로 일하고 있음을 보여준다.

이처럼 범위 검색 역시 **조건절에 사용되는 컬럼에 인덱스가 없다면** 풀 테이블 스캔을 피할 수 없다.
풀 테이블 스캔 상태에서 실제 쿼리를 실행해서 결과를 확인해보자.

쿼리 실행 결과

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

item_id	seller_id	item_name	category	price	stock_quantity
3	2	프리미엄 게이밍 마우스	전자기기	80000	200

6	5	고급 가죽 지갑	패션	70000	120
8	6	캐시미어 스웨터	패션	95000	70
14	2	인체공학 키보드	전자기기	90000	110
19	7	친환경 주방 세트	생활용품	60000	130

- 쿼리 실행 결과를 보면 `item_id` 순서로 정렬된 것을 확인할 수 있다.
- 테이블은 `item_id` 순서(테이블에 데이터가 물리적으로 저장된 순서)대로 정렬되어 있기 때문에 풀 테이블 스캔 과정에서 WHERE 조건을 만족한 순서대로 결과가 나온다.
- 하지만 이 순서를 데이터베이스가 보장하는 것은 아니다. 그냥 내부 실행 과정에 따라서 이 순서가 되었을 뿐이다. 만약 `item_id` 조건으로 정렬해야 한다면 `ORDER BY item_id` 를 추가하는 것을 권장한다.

이제 `price` 컬럼에 인덱스를 생성하고 실행 계획이 어떻게 변하는지 확인해 보자.

인덱스가 있을 때

이제 `items` 테이블의 `price` 컬럼에 인덱스를 생성하자.

```
CREATE INDEX idx_items_price ON items (price);
```

[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
200000	24
250000	7
300000	13
350000	23
450000	2
800000	20
1500000	10

- `price` 컬럼의 값 순서로 인덱스가 만들어진다.

이제 `EXPLAIN`으로 쿼리 실행 계획을 확인해 보자. 우리는 50,000원에서 100,000원 사이의 가격을 가진 상품들을 검색하는 쿼리를 사용해 볼 것이다.

```
EXPLAIN SELECT * 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

`price` 컬럼에 인덱스를 생성하자, 쿼리 실행 계획이 이전과는 완전히 다르게 매우 효율적으로 변경된 것을 확인할 수 있다.

- `type: range`: 가장 눈에 띄는 변화는 `type`이 `ALL`에서 `range`로 바뀐 점이다. 이는 데이터베이스가 **인덱스를 사용해 특정 범위의 데이터를 스캔**했음을 의미한다. 즉, `idx_items_price` 인덱스에서 `price`가 50000 이상인 지점을 찾은 뒤, 100000을 초과하는 지점이 나올 때까지만 순차적으로 인덱스를 읽었다는 뜻이다. 테이블 전체를 훑는 풀 테이블 스캔(`ALL`)과 비교할 수 없이 효율적인 방식이다.

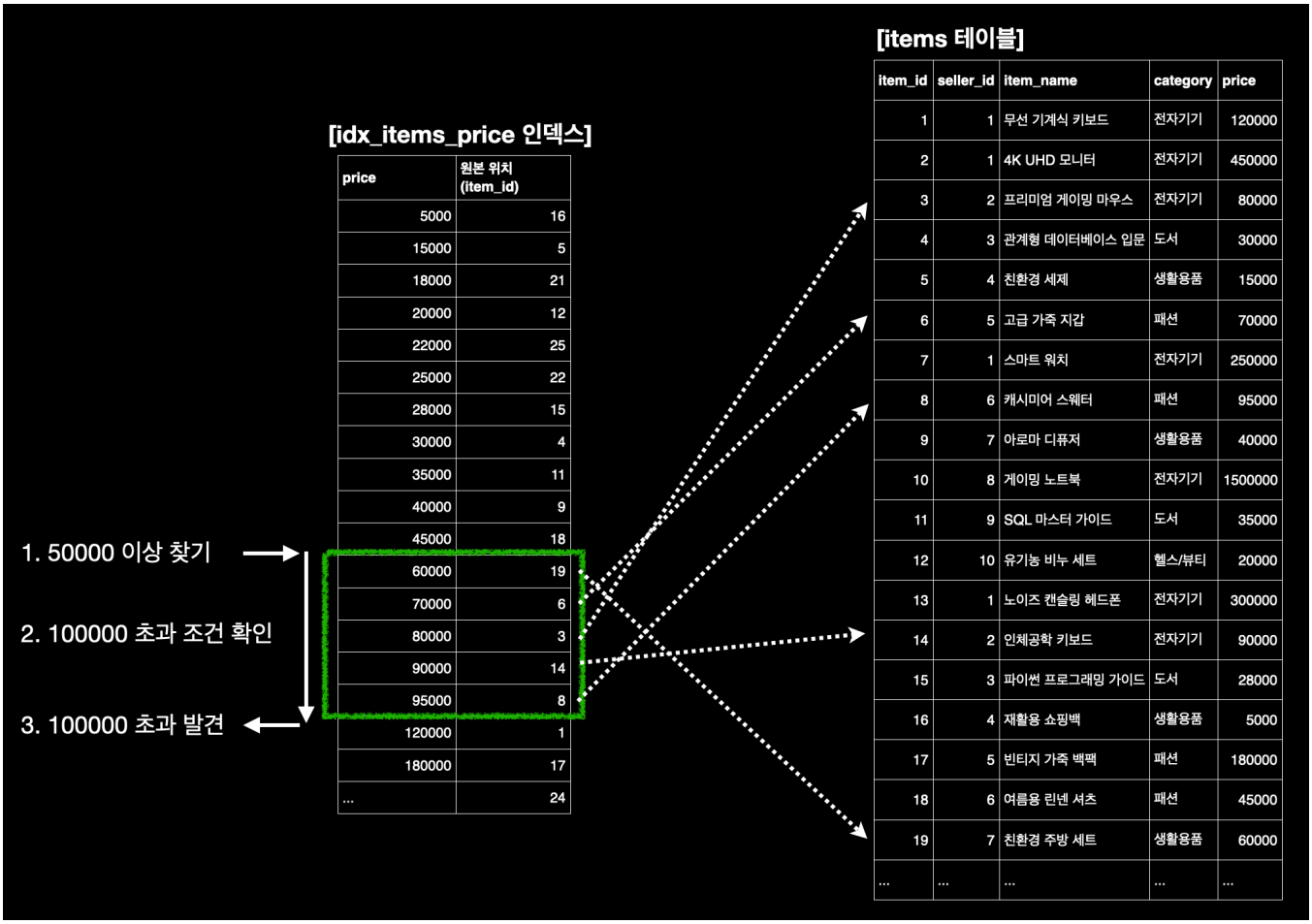
- `key: idx_items_price`: 쿼리 실행에 `idx_items_price` **인덱스가 사용되었음**을 명확히 보여준다. 옵티마이저는 `price` 컬럼에 대한 범위 검색에 이 인덱스를 사용하는 것이 가장 효율적이라고 판단한 것이다.

- `rows: 5`: 옵티마이저가 스캔할 것으로 예측하는 행의 수가 `5`로 크게 줄었다. 인덱스가 없을 때는 테이블 전체인 25개 행을 모두 스캔해야 했지만, 이제는 인덱스를 통해 조건에 맞는 5개의 데이터만 읽으면 된다는 것을 알고 있다. (예측치이다. 다를 수 있다.)

- `filtered: 100.00`: 인덱스를 통해서 스캔한 5개의 행을 100% 선택한다는 뜻이다.

- `Extra: Using index condition`: 이 부분도 중요한 최적화 정보다. 인덱스 정보만으로 `WHERE` 조건절을 최대한 필터링한 후, 조건을 만족하는 데이터의 전체 행만 가져왔다는 뜻이다.

인덱스 범위 검색 분석



인덱스의 범위 검색(`range`)는 매우 효율적으로 작동한다. 처음 한 번만 찾고, 이후에는 별도의 탐색 과정 없이 연속해서 결과를 구할 수 있다. 작동 순서를 확인해보자.

1. 먼저 인덱스의 `price` 항목에서 50000원 이상인 조건을 찾는다. 이 조건은 이진 탐색의 원리를 사용하므로 매우 빨리 찾을 수 있다. 여기서는 60000원을 찾는다.
2. `price`가 순서대로 정렬되어 있기 때문에 인덱스의 바로 다음 행으로 넘어가서 100000원을 초과했는지 확인한다. 그리고 인덱스의 다음 행으로 넘어가면서 이 과정을 반복한다.
    - 다음 행은 70000원이다. 조건에 부합하므로 결과의 대상이 된다.
    - 다음 행은 80000원이다. 조건에 부합하므로 결과의 대상이 된다.
    - 다음 행은 90000원이다. 조건에 부합하므로 결과의 대상이 된다.
    - 다음 행은 95000원이다. 조건에 부합하므로 결과의 대상이 된다.
    - 다음 행은 120,000원이다. 조건에 부합하지 않는다.
3. 100000원 초과 항목을 발견했으므로 탐색을 종료한다.

만약 이 인덱스가 없다면, 데이터베이스는 `items` 테이블 전체를 스캔하여 조건에 맞는 행을 찾아야 할 것이다. 인덱스가 범위 검색에서도 쿼리 성능을 크게 향상시킬 수 있다는 것을 알 수 있다.

쿼리 실행 결과

실제 쿼리를 실행해서 결과를 확인해 보자. 인덱스를 사용했을 때 결과는 어떻게 달라질까?

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

[실행 결과]

item_id	seller_id	item_name	category	price
19	7	친환경 주방 세트	생활용품	60000
6	5	고급 가죽 지갑	패션	70000
3	2	프리미엄 게이밍 마우스	전자기기	80000
14	2	인체공학 키보드	전자기기	90000
8	6	캐시미어 스웨터	패션	95000

여기서 주목할 점은 **결과가 `price` 순서로 정렬되었다는 것**이다. 가장 오른쪽의 `price` 컬럼을 확인해보자.

인덱스가 없을 때는 `item_id` 순서(테이블에 데이터가 물리적으로 저장된 순서)로 결과가 나왔지만, `idx_items_price` 인덱스를 사용한 후에는 **인덱스 키인 `price`를 기준으로 정렬**된 결과가 나왔다.

이는 데이터베이스가 `idx_items_price` 인덱스를 `price` 순서대로 스캔하면서 조건에 맞는 `item_id`를 찾고, 그 `item_id`를 사용해 원본 테이블에서 데이터를 가져왔기 때문이다. 이처럼 인덱스를 사용하면 쿼리 결과의 정렬 순서가 달라질 수 있다는 점을 알아두는 것이 중요하다.

하지만 이 순서를 데이터베이스가 항상 보장하는 것은 아니다. 그냥 내부 과정의 결과에 따라서 이 순서가 되었을 뿐이다. 만약 `price` 조건으로 정렬해야 한다면 `ORDER BY price` 를 추가하는 것을 권장한다.

인덱스와 LIKE 범위 검색

`LIKE` 절에서 인덱스를 사용하려면, 와일드카드(`%`)가 검색어의 뒤쪽에 위치해야 한다.

- `WHERE item_name LIKE '게이밍%'` : 인덱스 사용 가능 (O)

- `WHERE item_name LIKE '%게이밍'` : 인덱스 사용 불가 (X)
- `WHERE item_name LIKE '%게이밍%'` : 인덱스 사용 불가 (X)

`%`가 앞에 있으면 시작점이 불분명해져 정렬된 인덱스를 활용할 수 없기 때문이다. 예시로 살펴보자.

LIKE 검색 성공 예제: 와일드카드(`%`)가 뒤에 오는 경우

"상품 이름이 '게이밍'으로 시작하는 모든 상품을 찾아보자."
이 쿼리는 와일드카드(`%`)가 검색어 뒤에 붙어있으므로, `item_name`으로 정렬된 인덱스를 효율적으로 활용할 수 있다.

```sql
SELECT * FROM items WHERE item_name LIKE '게이밍%';
```

item_id	seller_id	item_name	category	price
10	8	게이밍 노트북	전자기기	1500000
24	2	게이밍 의자	전자기기	200000

실행 계획을 확인해보자.

```sql
EXPLAIN SELECT * FROM items WHERE item_name LIKE '게이밍%';
```

여기서도 인덱스의 범위 검색(`range`)을 사용할 수 있다. 왜냐하면 글자가 정렬되어 있기 때문이다. 처음 한 번만 찾고, 이후에는 별도의 탐색 과정 없이 연속해서 결과를 구할 수 있다. 작동 순서를 확인해보자.

1. 먼저 인덱스의 `item_name` 항목에서 `게이밍`으로 시작하는 조건을 찾는다. 이 조건은 이진 탐색의 원리를 사용하므로 매우 빨리 찾을 수 있다. 여기서는 게이밍 노트북을 찾는다.
2. `item_name`이 순서대로 정렬되어 있기 때문에 인덱스의 바로 다음 행으로 넘어가서 게이밍으로 시작하는지 확인한다. 그리고 인덱스의 다음 행으로 넘어가면서 이 과정을 반복한다.
    - 다음 행은 '게이밍 의자'이다. 게이밍으로 시작하는 조건에 부합하므로 결과의 대상이 된다.
    - 다음 행은 'SQL 마스터 가이드'이다. 게이밍으로 시작하는 조건에 부합하지 않는다.
3. '게이밍'으로 시작하지 않는 항목을 발견했으므로 탐색을 종료한다.

이것이 가능한 이유는 국어사전에서 '게'로 시작하는 단어를 찾는 것과 원리가 같기 때문이다. 쉽게 이야기해서 글자순으로 정렬되어 있기 때문이다. 데이터베이스는 `idx_items_item_name` 인덱스에서 '게이밍'으로 시작하는 첫 번째 위치를 빠르게 찾은 뒤, '게이밍'으로 시작하지 않는 단어가 나올 때까지 인덱스를 순차적으로 읽기만 하면 된다.

[실행 결과]

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

- `type: range`: `item_name`이 '게이밍'으로 시작하는 특정 범위를 인덱스에서 스캔했음을 의미한다. 이는 풀 테이블 스캔(`ALL`)보다 훨씬 효율적이다.
- `key: idx_items_item_name`: 우리가 생성한 `idx_items_item_name` 인덱스가 올바르게 사용되었다.

- `rows: 2`: 옵티마이저는 약 2개의 행만 스캔하면 결과를 찾을 수 있을 것으로 예측한다.

LIKE 검색 실패 예제: 와일드카드(`%`)가 앞에 오는 경우

"상품 이름에 '게이밍'이 포함된 모든 상품을 찾아보자."

이 쿼리는 실무에서 검색 기능을 만들 때 매우 흔하게 사용된다. 하지만 와일드카드(`%`)가 검색어 앞에 붙어있기 때문에 인덱스의 장점을 활용할 수 없다.

```
SELECT * FROM items WHERE item_name LIKE '%게이밍%';
```

item_id	seller_id	item_name	category	price
3	2	프리미엄 게이밍 마우스	전자기기	80000
10	8	게이밍 노트북	전자기기	1500000
24	2	게이밍 의자	전자기기	200000

- 프리미엄 게이밍 마우스도 추가로 포함되었다.

실행 계획을 분석해보자.

```
EXPLAIN SELECT * FROM items WHERE item_name LIKE '%게이밍%';
```

[실행 결과]

id	type	key	rows	filtered	Extra
1	ALL	NULL	25	11.11	Using where

실행 계획이 인덱스가 없던 시절로 돌아갔다.

- `type: ALL`: **풀 테이블 스캔**이 발생했다.
- `key: NULL`: `idx_items_item_name` 인덱스가 있음에도 불구하고 사용되지 않았다.

국어사전에서 **중간**에 '게이밍'이라는 글자가 들어간 단어를 찾으려면 어떻게 해야 할까? '가'부터 '힣'까지 모든 단어를 하나씩 다 훑어보는 수밖에 없다. 데이터베이스도 마찬가지다. `item_name`의 시작 글자를 알 수 없으므로, 정렬된 인

덱스는 아무런 도움이 되지 않는다. 결국 데이터베이스는 테이블의 모든 데이터를 처음부터 끝까지 스캔하는 최악의 방법을 선택하게 된다.

> ⚠ **LIKE 검색은 %를 마지막에 사용할 때만 인덱스를 활용할 수 있다.**
> `LIKE`도 범위 검색을 사용할 수 있다. 단 인덱스를 사용하려면 반드시 와일드카드(`%`)를 뒤에 사용해야 한다.

> ✨ **실무 팁: 전문 검색, 전체 문자 검색 (Full-Text Search)**
> 이처럼 `LIKE '%검색어%'` 방식은 데이터가 많아질수록 성능이 심각하게 저하되어 실제 서비스에서는 사용하기 어렵다.
>
> 이런 '내용 검색' 또는 '포함 검색' 문제를 해결하기 위해 데이터베이스는 **전문 검색(Full-Text Search)**이라는 특수한 기능을 제공한다. 전문 검색 인덱스는 B-Tree 인덱스와는 달리, 텍스트를 단어(토큰) 단위로 쪼개서 인덱싱하는 방식이다. 이를 통해 텍스트 중간에 있는 단어도 매우 빠르게 검색할 수 있다.
>
> 만약 쇼핑몰에서 상품명 검색 기능을 구현해야 한다면, `LIKE` 대신 `MATCH ... AGAINST` 구문을 사용하는 전문 검색 기능을 도입하는 것이 해결 방법이다. 필요하다면 'FullText Search'라는 키워드로 검색해보자.

인덱스와 정렬

데이터베이스에서 **정렬(`ORDER BY`) 작업은 생각보다 비용이 많이 드는 무거운 작업** 중 하나다. 왜냐하면 조건에 맞는 데이터를 모두 찾은 후에, 그 결과를 서로 비교하면서 순서에 맞게 다시 정렬해야 하기 때문이다. 찾은 데이터가 수십, 수백만 건이라면 이 데이터를 정렬 알고리즘을 사용해서 정렬해야 하는데, 이 정렬 과정에서 엄청난 부하가 발생할 수 있다.

하지만 우리에게는 인덱스가 있다. 인덱스는 이미 데이터가 특정 순서로 정렬된 자료구조다. 그렇다면 이 **정렬된 인덱스를 활용해서 `ORDER BY` 작업의 성능을 획기적으로 개선**할 수 있지 않을까?

`ORDER BY`가 인덱스를 잘 활용하면, 별도의 정렬 과정 없이 이미 정렬된 인덱스를 순서대로 읽기만 하면 되므로 매우 빠르게 동작한다. 데이터베이스는 이 과정에서 `filesort`**라는 별도의 정렬 작업을 생략**할 수 있게 된다. 여기서 `filesort`라는 이름만 보고 파일 시스템을 사용한다고 오해하면 안 된다. 실제로는 메모리나 디스크를 사용해 정렬하

는 내부 프로세스를 의미한다. 우리의 목표는 바로 이 비효율적인 `filesort`를 피하는 것이다.

인덱스를 활용하지 않는 다음 쿼리를 보면 `filesort`를 확인할 수 있다.

```
EXPLAIN SELECT * FROM items
ORDER BY stock_quantity;
```

id	table	type	key	rows	filtered	Extra
1	items	ALL	NULL	25	100.00	Using filesort

- Extra 항목에 `Using filesort`를 확인할 수 있다. 데이터를 모두 찾은 후에 `stock_quantity`를 기준으로 정렬 작업이 추가된다.

인덱스를 사용해 정렬까지 한 번에 처리하는 경우

가장 이상적인 상황은 `WHERE`절의 조건과 `ORDER BY`절의 정렬 기준이 같아서, **인덱스 하나로 검색과 정렬을 모두 해결**하는 경우다.

앞서 `price`로 범위 검색을 했던 쿼리에 `ORDER BY price`를 추가해서 실행 계획을 확인해 보자. 우리는 이미 `price` 컬럼에 `idx_items_price` 인덱스를 만들어 두었다.

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

[실행 결과]

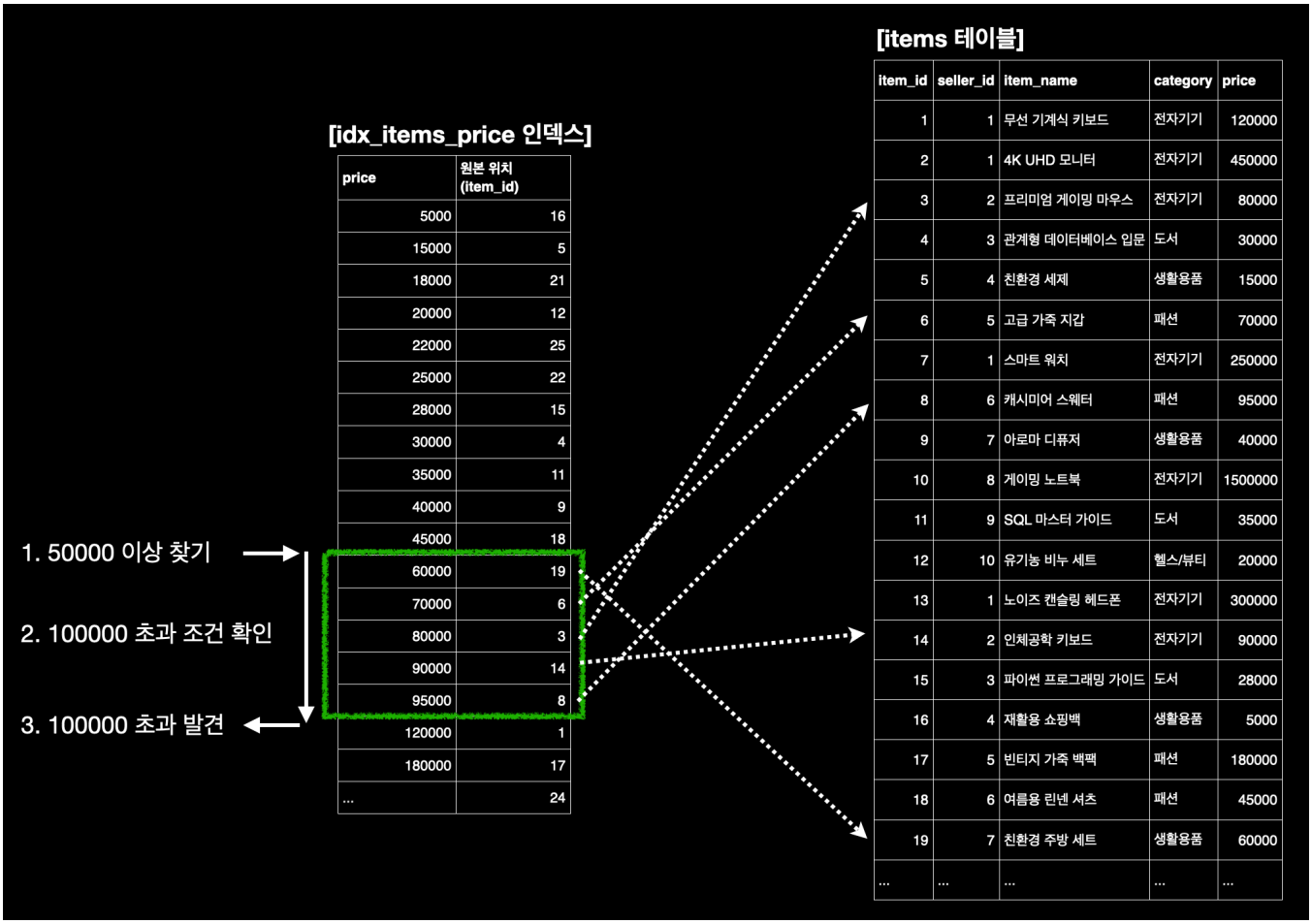
id	type	key	rows	filtered	Extra
1	range	idx_items_price	5	100.00	Using index condition

이 실행 계획을 자세히 분석해 보자.

- `type: range`: `idx_items_price` 인덱스를 사용해 특정 범위(50000 ~ 100000)를 효율적으로 스캔했다.
- `key: idx_items_price`: `idx_items_price` 인덱스가 사용되었다.

- `Extra: Using index condition`: `WHERE` 조건 필터링에 인덱스를 사용했다.

여기서 가장 중요한 것은 `Extra` 컬럼에 앞서 설명한 별도의 정렬 작업인 `Using filesort`**가 없다는 점**이다. 왜 없을까?



데이터베이스 옵티마이저는 `idx_items_price` 인덱스가 이미 `price` 순서로 정렬되어 있다는 사실을 알고 있다. 따라서 `WHERE` 조건에 맞는 데이터를 찾기 위해 인덱스를 스캔하는 것만으로도 자연스럽게 `price` 순서로 정렬된 결과를 얻을 수 있다. 즉, **별도의 정렬 작업을 할 필요가 전혀 없는 것**이다.

쿼리 실행 결과

`ORDER BY`를 제외하고 같은 쿼리를 실행해서 결과를 확인해 보자.

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

[실행 결과]

item_id	seller_id	item_name	category	price
19	7	친환경 주방 세트	생활용품	60000

6	5	고급 가죽 지갑	패션	70000
3	2	프리미엄 게이밍 마우스	전자기기	80000
14	2	인체공학 키보드	전자기기	90000
8	6	캐시미어 스웨터	패션	95000

결과가 `price` 순서로 정렬되어 있는 것을 확인할 수 있다. 가장 오른쪽 `price` 컬럼을 확인해 보라. `ORDER BY price`를 명시하지 않았음에도 인덱스를 스캔한 순서 덕분에 이미 결과가 정렬된 것이다.

이것이 인덱스를 활용한 `ORDER BY` 최적화의 핵심이다. `WHERE`절과 `ORDER BY`절이 동일한 인덱스를 효율적으로 사용할 수 있다면, 데이터베이스는 정렬을 생략하고, 가장 빠른 방식으로 쿼리를 처리한다.

인덱스를 역방향으로 조회하는 경우

`ORDER BY`를 사용할 때 항상 오름차순(`ASC`)으로만 정렬하는 것은 아니다. 쇼핑몰에서는 최신 상품이나 가격이 높은 상품을 먼저 보여주는 것처럼, 내림차순(`DESC`) 정렬도 매우 흔하게 사용된다.

그렇다면 `ORDER BY price DESC` 처럼 내림차순 정렬을 사용하면 `filesort`가 발생할까?

결론부터 말하면, **단일 컬럼 인덱스에서는 `filesort` 없이 효율적인 처리가 가능**하다. 데이터베이스 옵티마이저는 인덱스를 거꾸로 읽는, 즉 **역방향 스캔(Backward Index Scan)**을 할 수 있기 때문이다.

`price`가 비싼 순서대로 상품을 조회하는 쿼리의 실행 계획을 살펴보자.

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

[실행 결과]

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

실행 계획의 `Extra` 컬럼을 주목하자. `Using filesort`는 없지만, `Backward index scan`이라는 새로운 구문이 등장했다.

- `Backward index scan`: 옵티마이저가 `idx_items_price` 인덱스를 끝에서부터 앞으로, 즉 역순으로 스캔했음을 의미한다. 인덱스는 양방향 탐색이 가능하다. 따라서 `price`가 높은 값부터 낮은 값 순서로 인덱스를 탐색하여 `WHERE` 조건에 맞는 데이터를 찾는다. 이 과정에서 이미 정렬 순서가 만족되므로 별도의 `filesort` 작업이 필요 없다.

이처럼 인덱스를 역방향으로 스캔하는 것만으로도 `filesort`를 피할 수 있으므로 매우 효율적인 방식이다.

> **정방향 스캔이 역방향 스캔보다 미세하게 더 빠르다.**
> 정방향 인덱스 스캔이 미세하게 더 빠른 이유는 **컴퓨터 하드웨어의 '미리 읽기(Prefetching)' 기능 때문**이다.
> 컴퓨터는 데이터를 정방향(1, 2, 3...)으로 읽을 것을 예측하고, 다음 데이터를 미리 준비해 둔다. 이 방식으로 하드웨어가 최적화되어 있어 효율이 가장 높다.
> 하지만 이 성능 차이는 **미미해서 실무에서는 거의 무시해도 된다.** `ORDER BY`에서 `filesort`**를 피하는 것이 수백 배 더 중요하다.**

내림차순 인덱스 (Descending Index)

역방향 스캔은 효율적이지만, 여기서 한 걸음 더 나아가 정렬 방향과 일치하는 인덱스를 직접 만들어 줄 수도 있다.
MySQL 8.0 버전부터는 **내림차순 인덱스(Descending Index)** 생성을 정식으로 지원한다.

내림차순 인덱스는 데이터 자체를 처음부터 내림차순으로 정렬하여 저장하는 인덱스다.

기존 `idx_items_price` 인덱스를 삭제하고, `price`에 대한 내림차순 인덱스를 새로 만들어보자.

```
-- 기존 오름차순 인덱스 삭제
DROP INDEX idx_items_price ON items;

-- price 컬럼에 내림차순 인덱스 생성
CREATE INDEX idx_items_price_desc ON items (price DESC);
```

- `(price DESC)` 부분을 보자. 인덱스를 내림차순으로 정렬된 상태로 만드는 것이다.

이제 이 내림차순 인덱스가 있는 상태에서 다시 `ORDER BY price DESC` 쿼리의 실행 계획을 확인해 보자.

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

[실행 결과]

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

`key` 컬럼에서 우리가 새로 만든 `idx_items_price_desc` 인덱스가 사용된 것을 볼 수 있다. 가장 주목할 점은 `Extra` 컬럼에서 `Backward index scan`이 사라지고 `Using index condition`**만 남았다는 것**이다.

이는 옵티마이저가 더 이상 인덱스를 '거꾸로' 읽는 수고를 할 필요가 없어졌음을 의미한다. 쿼리가 요구하는 정렬 순서(`DESC`)와 인덱스의 정렬 순서(`DESC`)가 완벽하게 일치하므로, 인덱스를 자연스러운 순서(정방향)로 스캔하기만 하면 된다. 이것이 `ORDER BY` 절을 최적화하는 가장 이상적인 방법이다.

내용을 확인했다면 다음 과정을 위해 인덱스를 원래대로 다시 만들어두자.

```
-- 내림차순 인덱스 삭제
DROP INDEX idx_items_price_desc ON items;

-- price 컬럼에 오름차순 인덱스 생성
CREATE INDEX idx_items_price ON items (price);
```

- `(price)` 마지막에 DESC가 없어야 한다.

> ✨ **실무 팁**
> 단일 컬럼 인덱스에서는 역방향 스캔과 내림차순 인덱스 간의 성능 차이가 크지 않을 수 있다. 하지만 `ORDER BY category ASC, registered_date DESC`처럼 **여러 컬럼에 대해 서로 다른 정렬 순서(오름차순과 내림차순의 혼합)가 필요한 복잡한 쿼리**에서는 내림차순 인덱스의 진가가 발휘된다. 이런 경우, 정렬 순서에 맞춰 정확하게 생성된 다중 컬럼 인덱스(복합 인덱스)는 쿼리 성능을 극적으로 향상시킬 수 있다. 다중 컬럼 인덱스(복합 인덱스)는 뒤에서 알아본다.

정리

인덱스가 필요한 이유

- 데이터양이 많아질수록 인덱스 없는 검색은 **풀 테이블 스캔(Full Table Scan)**으로 인해 매우 느려진다.
- 풀 테이블 스캔은 테이블의 모든 데이터를 처음부터 끝까지 순차적으로 비교하는 방식으로, 데이터 양에 비례하여 성능이 저하된다.
- 서비스의 빠른 응답 속도는 사용자 경험에 매우 중요하며, 풀 테이블 스캔은 이를 저해하는 주된 원인이다.
- `WHERE` 절에 자주 사용되는 컬럼에 **인덱스**를 생성하는 것이 기본적인 해결책이다.

인덱스 소개

- 인덱스는 데이터베이스의 검색 성능을 향상시키는, 책의 '찾아보기(색인)'와 같은 자료 구조이다.
- 특정 컬럼의 값과 해당 데이터의 물리적 위치 주소를 쌍으로 저장한다.
- 인덱스의 핵심은 데이터가 **항상 정렬된 상태**로 유지된다는 점이며, 이를 통해 데이터를 매우 빠르게 찾을 수 있다.
- 검색 시 테이블 전체를 스캔하는 대신, 정렬된 인덱스를 먼저 탐색하여 원하는 데이터의 위치로 즉시 이동한다.

트리 자료 구조

- 인덱스는 주로 **트리(Tree) 자료 구조**, 특히 B-Tree를 사용하여 구현된다.
- 이진 탐색 트리와 같이 데이터를 정렬된 상태로 저장하여, 검색 시 `O(log n)`의 매우 빠른 시간 복잡도를 가진다.
- 데이터가 100만 건이라도 약 20번의 비교만으로 데이터를 찾을 수 있다.
- 대부분의 데이터베이스는 트리의 한쪽으로 치우치는 문제를 해결한 **벨런스 트리(Balanced Tree)**를 사용하여 최악의 경우에도 `O(log n)` 성능을 보장한다.

인덱스 생성, 조회, 삭제

- `CREATE INDEX`, `SHOW INDEX`, `DROP INDEX` 명령어를 사용하여 인덱스를 생성, 조회, 삭제한다.
- MySQL에서는 `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE` 제약조건을 설정하면 해당 컬럼에 인덱스가 자동으로 생성된다.
- `EXPLAIN` 명령어를 쿼리 앞에 붙이면 데이터베이스의 실행 계획을 확인할 수 있으며, 이를 통해 인덱스 사용 여부를 반드시 점검해야 한다.
- 실행 계획의 `type`이 `ALL`이면 풀 테이블 스캔, `ref`나 `range`이면 인덱스 사용을 의미한다.

인덱스와 동등 비교

- 동등 비교(`=`) 조건에 인덱스가 사용될 때, 실행 계획의 `type`은 `ref`로 표시된다.
- `ref`는 인덱스를 통해 조건에 맞는 데이터를 매우 효율적으로 참조했다는 의미이다.
- 인덱스를 사용하면 탐색해야 할 행(`rows`)의 수가 극적으로 줄어들어 검색 성능이 크게 향상된다.

인덱스와 범위 검색

- `BETWEEN`, `>`, `<`와 같은 범위 검색에 인덱스가 사용될 때, 실행 계획의 `type`은 `range`로 표시된다.
- 데이터베이스는 정렬된 인덱스에서 범위의 시작점을 빠르게 찾고, 범위가 끝날 때까지만 순차적으로 스캔하므로 효율적이다.
- 인덱스를 사용하면 결과가 인덱스 키 순서로 정렬되어 나올 수 있지만, 정확한 정렬을 위해서는 `ORDER BY`를 명시해야 한다.

인덱스와 LIKE 범위 검색

- `LIKE` 검색에서 인덱스를 활용하려면 와일드카드(`%`)가 반드시 검색어의 **뒤쪽**에 위치해야 한다 (`LIKE '검색어%'`).
- 와일드카드가 검색어의 앞에 오면 (`LIKE '%검색어'`) 시작점을 특정할 수 없어 인덱스를 사용하지 못하고 풀 테이블 스캔이 발생한다.
- 텍스트 중간에 포함된 단어를 검색하기 위해서는 `LIKE` 대신 **전문 검색(Full-Text Search)** 기능을 사용해야 한다.

인덱스와 정렬

- 정렬(`ORDER BY`)은 비용이 많이 드는 작업이지만, 인덱스를 활용하면 별도의 정렬 과정(`filesort`)을 생략하여 성능을 개선할 수 있다.
- `WHERE` 조건과 `ORDER BY` 조건이 인덱스와 일치하면, 데이터베이스는 이미 정렬된 인덱스를 순서대로 읽기만 하므로 `filesort`가 발생하지 않는다.
- 내림차순 정렬(`DESC`) 시에는 인덱스를 거꾸로 읽는 **역방향 스캔(Backward index scan)**을 통해 `filesort`를 피할 수 있다.
- 쿼리의 정렬 순서와 일치하는 **내림차순 인덱스**(`(컬럼명 DESC)`)를 생성하면 가장 이상적인 정렬 최적화가 가능하다.