

# RealMySQL8.0 - Chapter8

인덱스는 데이터베이스 쿼리의 성능을 언급하면서 빼놓을 수 없는 부분이다.

-> 맞음

## △ 학습 목표

쿼리의 개발 튜닝을 설명하기 전 MySQL에서 사용 가능한 인덱스의 종류 및 특성을 학습함

인덱스의 특성과 차이는 물리 수준의 모델링을 할 때도 중요한 요소가 된다.

저자는 인덱스에 대한 기본 지식은 쿼리 튜닝의 기본이 된다고 설명함.

## 디스크 읽기 방식

사용된 키워드

- 랜덤 I/O
- 순차 I/O

컴퓨터의 CPU나 메모리처럼 전기적 특성을 띤 장치의 성능은 짧은 시간 동안 매우 빠른 속도로 발전했지만 디스크 같은 기계식 장치의 성능은 상당히 제한적으로 발전했다.

최근에는 자기 디스크 원판에 의존하는 하드 디스크가 아닌 SSD 드라이브가 많이 활용되고 있지만, 여전히 데이터 저장 매체는 컴퓨터에서 가장 느린 부분이라는 사실에는 변함이 없다

데이터베이스의 성능 튜닝은 어떻게 디스크 I/O를 줄이느냐가 관건일 대가 상당히 많다.

- 데이터 저장 매체는 컴퓨터에서 가장 느린 부분이기 때문
- 전기적 특성을 띤 장치인 CPU, 메모리는 빠르게 발전했지만 디스크 같은 기계식 장치는 제한적으로 발전함.
- SSD가 나왔지만 그래도 느린 편임

## 하드 디스크 드라이브(HDD) 와 솔리드 스테이트 드라이브(SSD)

컴퓨터에서 CPU나 메모리 같은 주요 장치는 대부분 전자식 장치지만 하드 디스크 드라이브는 기계식 장치다. 그래서 데이터베이스 서버에서는 항상 디스크 장치가 병목이 된다.

-> 기계식 장치이기 때문에 병목이 발생하는 건가요?

-> 이러한 기계식 하드 디스크 드라이브를 대체하기 위해 전자식 저장 매체인 SSD가 많이 출시되고 있습니다.

SSD는 기존 하드 디스크 드라이브에서 데이터 저장용 플래터(원판)를 제거하고 그 대신 플래시 메모리를 장착하고 있습니다.

- 원판을 회전시킬 필요가 없기 때문에 데이터를 빠르게 읽고 쓸 수 있다.
- 물리적인 삭제가 발생하지 않는다. ( 이거 맞나? )
- 컴퓨터 메모리 보다 느리지만 기계식 하드 디스크 드라이브보다는 훨씬 빠르다.

디스크의 헤더를 움직이지 않고 한 번에 많은 데이터를 읽는 순차 I/O에서는 SSD가 하드 디스크 드라이브보다 조금 빠르거나 거의 비슷한 성능을 보이기도 한다.

하지만 SSD의 장점은 기존 하드 디스크 드라이브보다 랜덤 I/O가 훨씬 빠르다는 것이다.

데이터베이스 서버에서 순차 I/O 작업은 그다지 비중이 크지 않고 랜덤 IO를 통해 작은 데이터를 읽고 쓰는 작업이 대부분이므로 SSD의 장점은 DBMS용 스토리지에 최적이라고 볼 수 있다.

일반적인 웹 서비스 환경의 데이터베이스에서는 SSD가 하드 디스크드라이브 보다는 훨씬 빠르다.

## 랜덤 I/O와 순차 I/O

<https://hudi.blog/storage-and-random-sequential-io/>

랜덤 I/O라는 표현은 하드 디스크 드라이브의 플래터를 돌려서 읽어야 할 데이터가 저장된 위치로 디스크 헤더를 이동시킨 다음 데이터를 읽는 것을 의미합니다.

순차 I/O도 디스크 헤더를 이동시켜 데이터를 읽는 방식은 동일하다.

순차 I/O는 3개의 페이지( $3 \times 16\text{kb}$ )를 디스크에 기록하기 위해 1번 시스템 콜을 요청했지만, 랜덤 I/O는 3개의 페이지 디스크에 기록하기 위해 3번 시스템 콜을 요청했다.

즉, 디스크에 기록해야 할 위치를 찾기 위해 순차 I/O는 디스크의 헤드를 1번 움직였고, 랜덤 I/O는 디스크 헤드를 3번 움직였다.

-> 디스크에 데이터를 쓰고 읽는 데 걸리는 시간은 디스크 헤더를 움직여서 읽고 쓸 위치로 옮기는 단계에서 결정된다

그래서 순차 I/O는 랜덤 I/O보다 3배 빠르다고 할 수 있다.

-> 디스크의 성능은 디스크의 헤더의 위치 이동 없이 얼마나 많은 데이터를 한 번에 기록하느냐에 의해 결정된다고 볼 수 있다.

-> 여러 번 쓰기 또는 읽기를 요청하는 랜덤 I/O 작업이 작업 부하가 훨씬 더 크다.

-> 이런 문제점을 최소한으로 하기 위해 그룹 커밋, 바이너리 로그 버퍼, InnoDB 로그 버퍼 등의 기능이 내장돼 있다.

## 인덱스

책을 보면 맨끝에 색인이 존재합니다. 책에서 필요한 내용을 쉽게 찾기 위한 장치로 초성으로 정렬돼 있는 것이 특징입니다.

마찬가지로 DBMS의 인덱스는 데이터베이스 테이블의 데이터 검색을 쉽게 하기 위한 장치로 컬럼의 값과 해당 레코드가 저장된 주소를 키와 값의 쌍으로 만들어집니다.

인덱스의 특징을 보기 앞서 다음 세 자료구조를 봄야 합니다.

- SortedList

- DBMS의 인덱스와 같은 자료 구조입니다.
- 저장되는 값을 항상 정렬된 상태로 유지하는 자료 구조
- **ArrayList**
  - **ArrayList**는 데이터 파일과 같은 자료구조로 사용됩니다.
  - 값을 저장되는 순서 그대로 유지하는 자료 구조

**SortedList**의 장단점을 통해 인덱스의 장단점을 확인해보겠습니다.

- 단점
  - 데이터가 저장될 때마다 항상 값을 정렬해야 하므로 저장하는 과정이 복잡하고 느립니다. (**SortedList**)
  - 인덱스가 많은 테이블은 **INSERT** **UPDATE** **DELETE** 문장의 처리가 느려집니다. (인덱스)
- 장점
  - 이미 정렬된 상태이기 때문에 원하는 값을 빠르게 찾아올 수 있습니다. (**SortedList**)
  - 이미 정렬된 표(인덱스)를 가지고 있기 때문에 **SELECT** 문장은 매우 빠르게 처리할 수 있습니다. (인덱스)

결국 DBMS에서 인덱스는 데이터의 저장 성능을 희생하는 대신 데이터의 읽기 속도를 높이는 기능입니다. 그렇기 때문에 테이블의 인덱스 추가를 고려 할 때 데이터의 저장 속도를 어디까지 희생할지, 읽기 속도를 얼마나 더 빠르게 할지 결정해야 합니다.

또한 **SELECT** 쿼리의 **WHERE** 조건절에 사용하는 컬럼이라고 해서 전부 인덱스로 생성하는 것은 데이터 저장 성능이 떨어지고 인덱스의 크기가 비대해지는 문제점이 발생합니다.

△ 이 책에서는 키라는 말과 인덱스는 같은 의미로 사용함

인덱스는 데이터를 관리하는 방식, 중복 값 허용 여부에 등에 따라 여러 가지로 나눌 수 있습니다.  
먼저 인덱스를 역할별로 구분하면 **프라이머리 키**와 **보조 키**로 구분할 수 있습니다.

- 프라이머리 키 (Primary Key)
- 보조 키 (Secondary Key)

데이터 저장 방식(알고리즘)별로 구분할 경우 **B-Tree** 인덱스와 **Hash** 인덱스로 구분할 수 있습니다.

- B-Tree 알고리즘
- Hash 인덱스 알고리즘

데이터의 중복 허용 여부로 분류하면 유니크 인덱스와 유니크하지 않은 인덱스로 구분할 수 있다.

- 유니크 인덱스(Unique index)
- 유니크 하지 않은 인덱스 (Non-Unique index)

그 외 기능 인덱스의 기능별로 분류하면 전문 검색용 인덱스, 공간 검색용 인덱스 등이 있습니다.

# B-Tree 인덱스

B-Tree는 데이터베이스 인덱싱 알고리즘 가운데 가장 일반적이며 현재 가장 범용적인 목적으로 사용되는 인덱스 알고리즘입니다.

DBMS에서 사용하는 인덱스 알고리즘은 B-Tree를 변형한 B+-Tree, B\*-Tree(B star tree)가 있습니다.

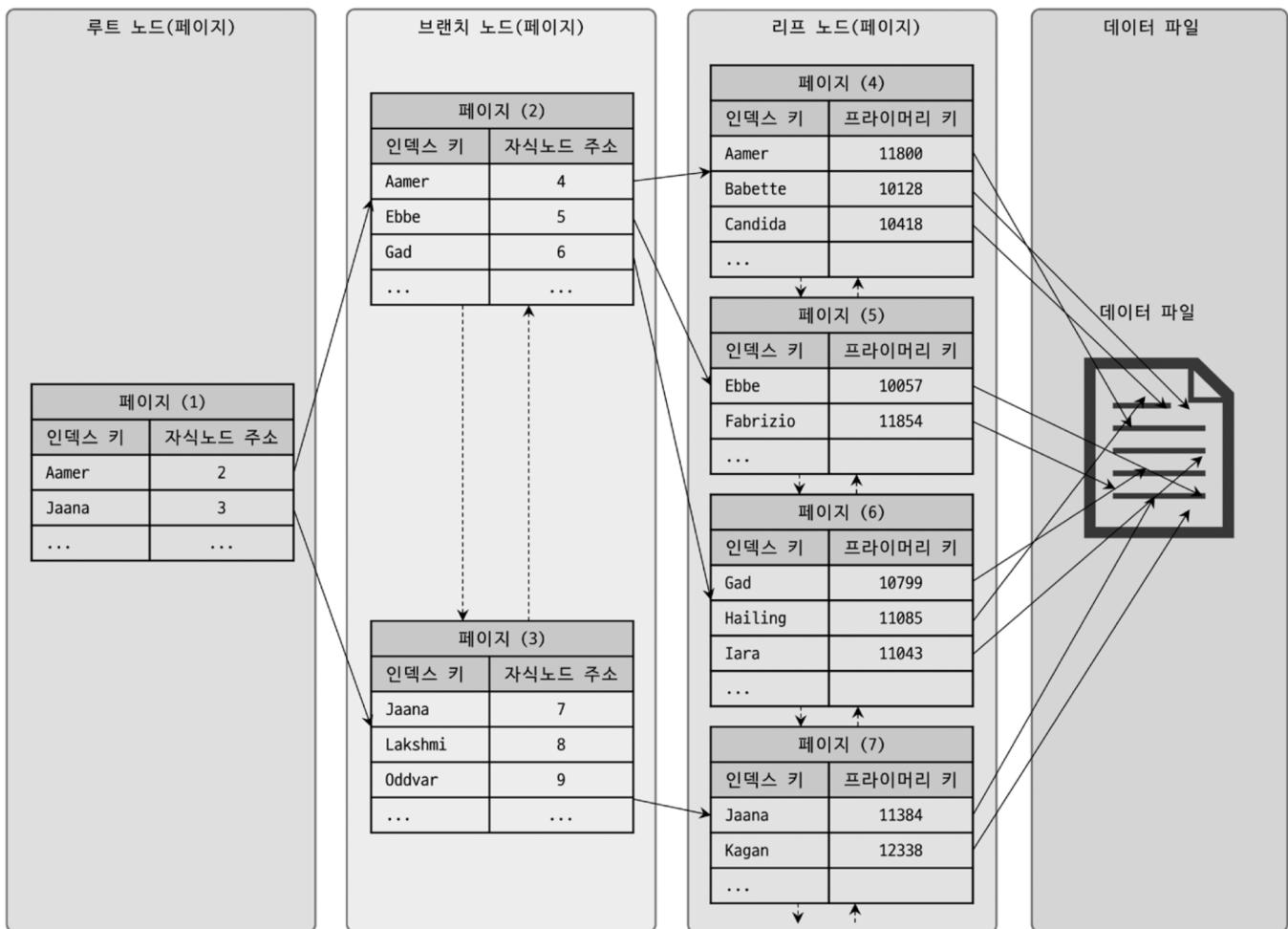
B-Tree는 컬럼의 원래 값을 변형시키지 않고 인덱스 구조체 내에서는 항상 정렬된 상태로 유지한다.

## 구조 및 특성

B-Tree는 트리 구조로 최상위에 하나의 루트 노드가 존재하고 하위에 자식 노드가 붙어 있는 형태입니다.

트리 구조에서 가장 하위에 있는 노드를 리프 노드라 하고, 루트 노드도 아니며 리프 노드도 아닌 중간의 노드를 브랜치 노드라고 합니다.

인덱스의 리프 노드는 항상 실제 데이터 레코드를 찾아가기 위한 주솟값을 가지고 있습니다.



### ▲ 참고

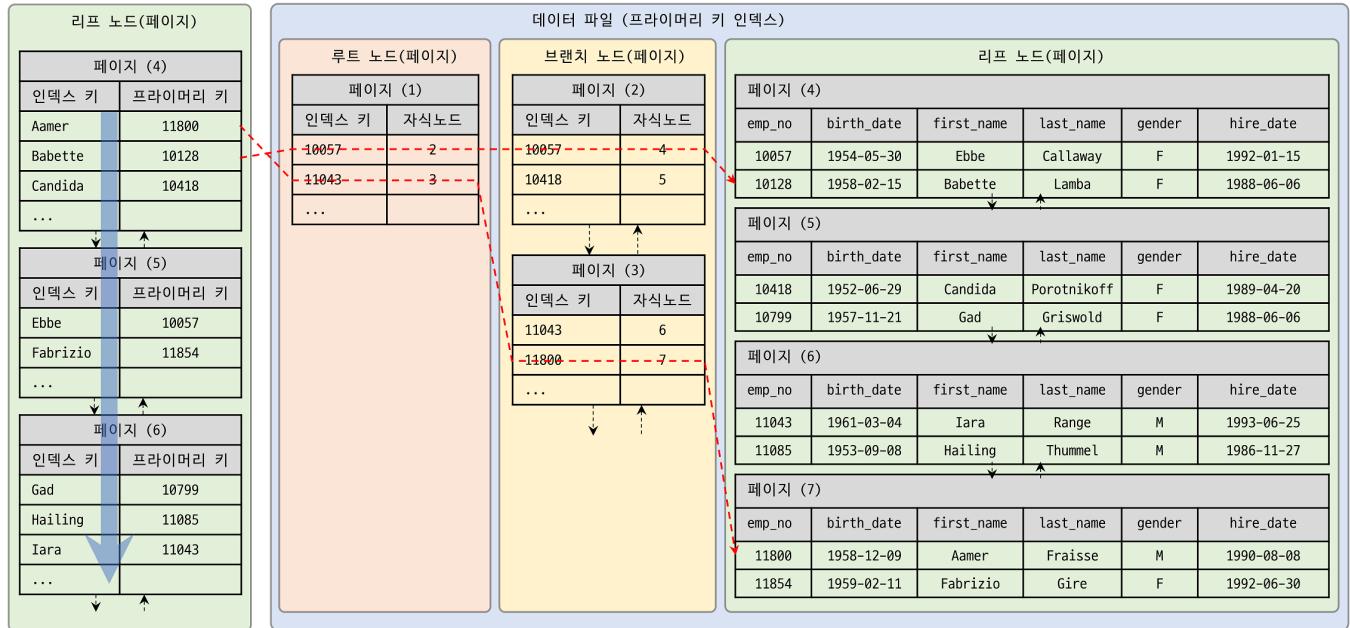
데이터 파일의 레코드는 Insert된 순서대로 저장되는 것은 아니다.

만약 테이블의 레코드를 전혀 삭제하거나 변경하지 않고 INSERT만 수행한다면 맞을 수 있도 있다.

하지만 레코드가 삭제되어 빈 공간이 생기면 그다음의 INSERT는 가능한 한 삭제된 공간을 재활용하도록 DBMS가 설계되기 때문에 항상 INSERT된 순서로 저장되는 것은 아니다.

인덱스의 키 값으로 모두 정렬되어 있지만 데이터 파일의 레코드는 정렬되지 않고 임의의 순서로 저장돼 있다.

- 인덱스는 테이블의 키 컬럼만 가지고 있으므로 나머지 컬럼을 읽으려면 데이터 파일에서 해당 레코드를 찾아야 한다.
- 인덱스의 리프 노드는 데이터 파일에 저장된 레코드의 주소를 가진다.



InnoDB 스토리지 엔진에서 가지는 B-Tree의 특징은 다음과 같습니다.

- InnoDB 스토리지 엔진을 사용하게 되면 테이블에서는 프라이머리 키가 ROWID의 역할을 한다.
- InnoDB 테이블은 프라이머리 키를 주소처럼 사용하기 때문에 논리적인 주소를 가진다고 볼 수 있습니다.
- InnoDB 테이블에서 인덱스를 통해 레코드를 읽을 때는 인덱스에 저장돼 있는 프라이머리 키 값을 이용해 프라이머리 키 인덱스를 한번 더 검색한 후, 프라이머리 키 인덱스의 리프 페이지에 저장돼 있는 레코드를 읽는다.
  - InnoDB 스토리지 엔진에서는 모든 세컨더리 인덱스 검색에서 데이터 레코드를 읽기 위해서는 반드시 프라이머리 키를 저장하고 있는 B-Tree를 다시 한번 검색해야 한다.

### 8.8 클러스트링 인덱스에서 살펴보기

## B-Tree 인덱스 키 추가 및 삭제

### △ 학습 목표

- 테이블의 레코드를 저장하거나 변경하는 경우 인덱스 키 추가나 삭제 작업이 발생한다.
- 인덱스 키 추가나 삭제가 어떻게 처리되는지 알아두면 쿼리의 성능을 쉽게 예측할 수 있을 것이다.

## 인덱스 키 추가

B-tree에 새로운 키 값이 저장될 때 테이블의 스토리지 엔진에 따라 새로운 키 값이 즉시 인덱스에 저장될 수도 있고 그렇지 않을 수도 있다.

-> 스토리지 엔진에 따라 B-tree에 키가 저장되는 방식이 다름

1. 새로운 키를 저장할 때 B-Tree에 적절한 위치를 검색한다.
2. 저장될 위치가 결정되면 키 값, 대상 레코드의 주소 정보를 리프 노드에 저장한다.
3. 리프 노드가 가득 찬 경우 리프 노드가 분리돼야 하는데, 상위 브랜치 노드까지 처리 범위가 넓어진다.

이런 작업 특징으로 B-Tree는 새로운 키를 추가하는 비용이 많이 듭니다.

### b-tree 삽입 과정에 대한 링크

📌 인덱스 추가로 인해 INSERT, UPDATE 문장이 어떤 영향을 받을까?

단순하게 계산할 수 있는 것이 아닌 테이블의 컬럼 수, 컬럼의 크기, 인덱스 컬럼의 특성을 파악해야 어떻게 영향을 주는지 알 수 있습니다.

대략적으로 계산하는 방법은 테이블에 레코드를 추가하는 작업 비용을 1로 가정한 다음 테이블의 인덱스에 키를 추가하는 작업 비용을 1.5정도로 예측할 수 있습니다.

만약 테이블에 인덱스가 3개라면 테이블에 인덱스가 하나도 없는 경우 작업비용은 1입니다. 만약 3개 인 경우는 5.5 정도의 비용이 발생하는데 아래의 계산 식을 참고하면 됩니다.

$$1.5 * 3 + 1$$

$$1.5 \times n + 1 \quad (n \geq 2) \quad (\text{테이블의 인덱스 개수 } n)$$

중요한 것은 대부분의 비용이 메모리와 CPU에서 처리하는 시간이 아닌 디스크로부터 인덱스 페이지를 읽고 쓰기를 해야 해서 걸리는 시간입니다.

참고로 InnoDB 스토리지 엔진이 아닌 경우 테이블에서는 INSERT 문장이 실행되면 즉시 새로운 키 값을 B-Tree인덱스에 변경합니다.

InnoDB는 이 작업을 프라이머리, 유니크 인덱스의 경우 B-Tree에 즉시 추가 삭제를 하지만 그 이외에는 키 추가 작업을 지연시켜 나중에 처리하도록 합니다.

- 체인지 버퍼 참고

## 인덱스 키 삭제

B-Tree의 키 값 삭제는 해당 키 값이 저장된 리프노드를 찾아서 삭제 마크만 하면 작업이 완료된다.

삭제 마킹된 인덱스 키 공간은 계속 그대로 방치하거나 재활용할 수 있다.

키 삭제로 인한 마킹 작업 또한 디스크 쓰기가 필요하므로 이 작업 역시 디스크 I/O가 필요한 작업이기 때문에 InnoDB 스토리지 엔진에서는 이 작업에 대한 버퍼링으로 지연 처리를 지원한다.

(참고: 처리가 지연된 인덱스 키 삭제 또한 사용자에게는 특별한 악영향 없이 MySQL 서버가 내부적으로 처리하므로 특별히 걱정할 것은 없다.)



```

if (next_page_no != FIL_NULL) {
    buf_block_t *next_block =
        btr_block_get(page_id_t(space, next_page_no), page_size,
RW_X_LATCH,
                      UT_LOCATION_HERE, index, mtr);

    page_t *next_page = buf_block_get_frame(next_block);
#ifdef UNIV_BTR_DEBUG
    ut_a(page_is_comp(next_page) == page_is_comp(page));
    ut_a(btr_page_get_prev(next_page, mtr) ==
page_get_page_no(page));
#endif /* UNIV_BTR_DEBUG */

    btr_page_set_prev(next_page,
buf_block_get_page_zip(next_block),
                      prev_page_no, mtr);
}
}

```

## 인덱스 키 변경

인덱스의 키 값은 그 값에 따라 저장될 리프 노드의 위치가 결정되므로 B-Tree의 키 값이 변경되는 경우에는 단순히 인덱스상의 키 값만 변경하는 것은 불가능하다.

B-Tree의 키 값 변경 작업은 먼저 키 값을 삭제한 후, 다시 새로운 키 값을 추가하는 형태로 처리된다.

결국 인덱스 키 값을 변경하는 작업은 기존 인덱스 키 값을 삭제한 후 새로운 인덱스 키 값을 추가하는 작업으로 처리되고, InnoDB 스토리지 엔진을 사용하는 테이블에 대해서는 이 작업 모두 체인지 버퍼를 활용해 지연 처리 될 수 있다.

## 인덱스 키 검색

**INSERT, UPDATE, DELETE** 작업을 할 때 인덱스 관리에 따르는 추가 비용을 감당하면서 인덱스를 구축하는 이유는 빠른 검색을 위해서입니다.

인덱스를 검색하는 과정을 트리 탐색이라고 하는데 B-Tree의 루트 노드 -> 브랜치 노드 -> 리프노드 순으로 이동하며 비교작업을 거칩니다. 이런 인덱스 트리 탐색은 **SELECT**뿐만 아니라 **UPDATE, DELETE**를 처리하기 위해 사용하기도 합니다.

B-Tree인덱스를 이용한 검색을 이용할 때 주의할점이 있습니다.

1. 인덱스를 이용한 검색은 100%일치 또는 값의 앞부분만 일치하는 경우에 사용할 수 있다.

2. 부등호 비교 조건에서도 사용가능하지만 인덱스를 구성하는 키 값의 뒷부분만 검색하는 용도로는 인덱스를 사용할 수 없다.
3. 인덱스를 이용한 검색에서 중요한 사실은 인덱스의 키 값에 변형이 가해진 후 비교되는 경우에는 절대 B-Tree의 빠른 검색을 사용할 수 없다.
  - 이미 변형된 값은 B-Tree 인덱스에 존재하는 값이 아니다. 따라서 함수나 연산을 수행한 결과로 정렬한다거나 검색하는 작업은 B-Tree의 장점을 이용할 수 없으므로 주의해야 한다.

InnoDB 테이블에서 지원하는 레코드 잠금, 넥스트 갭락이 검색을 수행한 인덱스를 접근 후 테이블의 레코드를 잡그는 방식으로 구현돼 있습니다.

그렇기에 DELETE, UPDATE의 잘못된 인덱스 설계는 테이블의 모든 레코드를 잠금 수 있기에 스토리지 엔진에서는 그만큼 인덱스의 설계가 중요하고 많은 부분에 영향을 미칩니다.

## B-Tree 인덱스 사용에 영향을 미치는 요소

B-Tree 인덱스는 인덱스를 구성하는 컬럼의 크기와 레코드의 건수, 그리고 유니크한 인덱스 키 값의 개수 등에 의해 검색이나 변경 작업의 성능이 영향을 받는다.

### 인덱스 키 값의 크기

InnoDB 스토리지 엔진은 디스크에 데이터를 저장하는 가장 기본 단위를 페이지(Page) 혹은 블록(Block)이라고 하며 이는 디스크의 모든 읽기 및 쓰기 작업의 최소 작업 단위가 됩니다.

또한 페이지는 InnoDB 스토리지 엔진의 버퍼풀에서 데이터를 버퍼링하는 기본 단위기도 합니다. 인덱스 또한 페이지 단위로 관리되며, 루트, 브랜치, 리프 노드를 구분한 기준이 페이지 단위입니다

```
#ifdef UNIV_DEBUG
struct Index_details {
    void add_page(const page_no_t page_no, const size_t level) {
        if (level < MAX_LEVEL) {
            m_pages[level].push_back(page_no);
        }
    }
    static const size_t MAX_LEVEL = 20;
    std::vector<page_no_t> m_pages[MAX_LEVEL];
};
#endif /* UNIV_DEBUG */
```



실제로 btr0btr 파일의 코드를 보면 page라는 단어를 많이 사용하는 것을 알 수 있음

이진(Binary) 트리는 각 노드가 자식 노드를 2개만 가지는 DBMS의 B-Tree가 이진 트리라면 인덱스 검색이 상당히 비효율적일 것이다.

- 깊은 트리 구조:** 이진 트리는 각 노드가 최대 두 개의 자식을 가질 수 있기 때문에, 동일한 수의 요소를 저장할 때 B-트리에 비해 더 깊은 구조를 가집니다. 따라서, 검색, 삽입, 삭제 등의 작업을 위해 더 많은 노드를 방문해야 하며, 이는 데이터베이스에서 더 많은 디스크 I/O를 발생시킵니다.
- 디스크 I/O 최적화 부족:** 데이터베이스 시스템에서는 디스크 I/O 작업이 성능에 큰 영향을 미칩니다. 이진 트리는 노드당 데이터가 적기 때문에, 같은 양의 데이터를 읽기 위해 더 많은 디스크 액세스가 필요할 수 있습니다. 반면 B-트리는 노드당 더 많은 데이터를 저장할 수 있어 디스크 I/O를 줄일 수 있습니다.
- 균형 유지의 어려움:** 일반적인 이진 트리는 데이터 삽입 순서에 따라 편향될 수 있으며, 이는 검색 성능을 저하시킬 수 있습니다. 균형 이진 트리(AVL 트리, 레드-블랙 트리 등)는 균형을 유지하도록 설계되었지만, 균형을 맞추기 위한 추가적인 연산이 필요합니다.
- 범위 쿼리의 비효율성:** 이진 트리는 특정 범위의 데이터를 검색하는 쿼리에 비효율적일 수 있습니다. B-트리는 순차적인 데이터 접근이 용이하여 범위 쿼리에 더 적합합니다.
- 메모리 사용의 비효율성:** 이진 트리는 노드당 정보를 적게 저장하기 때문에, 동일한 양의 데이터를 저장하기 위해 더 많은 메모리 오버헤드가 발생할 수 있습니다. B-트리는 노드당 더 많은 키를 저장할 수 있어 상대적으로 메모리 사용이 효율적입니다.

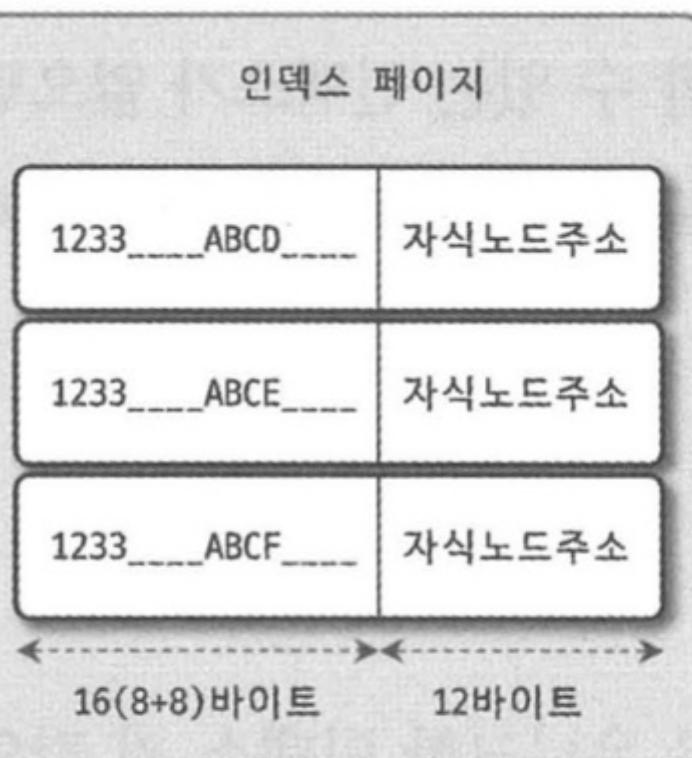


그림 8.7 인덱스 페이지의 구성

하나의 인덱스 페이지(16KB)에는 585개의 키를 저장할 수 있습니다. 즉 자식 노드를 585개를 가질 수 있는 B-Tree를 의미합니다.

인덱스 키 값이 커지는 경우 키 값의 크기가 두 배인 32바이트로 늘어났다고 가정하면 한 페이지에 인덱스 크를  $16*1024/(32 + 12) = 372$ 개를 저장할 수 있습니다.

SELECT쿼리가 레코드 500개를 읽어야 한다면 전자는 인덱스 페이지 한번으로 해결할 수 있지만 후자의 경우는 2번 이상 디스크를 읽어야 하기 때문에 그만큼 느려지는 것을 의미합니다

## B-Tree의 깊이

B-Tree 인덱스의 깊이는 상당히 중요하지만 직접 제어할 방법은 없습니다.

인덱스의 B-Tree 깊이가 3인 경우 키 값이 16바이트로 가정하면 최대 2억(585 585 585)개의 키를 담을 수 있지만 키 값이 32바이트로 늘어나면 5천만개로 줄어듭니다.

같은 양의 데이터를 다루게 된다면 그만큼 깊이가 늘어나고 디스크 읽기가 더 많이 필요하게 됩니다.

즉 인덱스 페이지의 크기를 늘리는 것은 권장하지 않습니다.

## 선택도 (기수성)

인덱스에서 선택도(Selectivity) 혹은 기수성(Cardinality)은 거의 같은 의미로 사용되며, 모든 인덱스 키 값 가운데 유니크한 값의 수를 의미합니다.

예를 들면 전체 인덱스 키 값이 100개인데 유니크한 값의 수는 10개라면 기수성은 10입니다.

인덱스 키 값 가운데 중복된 값이 많아지면 많아질수록 기수성은 낮아지고 동시에 선택도 또한 떨어집니다. 반대로 선택도가 높을 수록 검색 대상이 줄어들기 때문에 그만큼 빠르게 처리됩니다.

### ▲ 참고

선택도가 좋지 않다고 하더라도 정렬이나 그룹핑과 같은 작업을 위해 인덱스를 만드는 것이 훨씬 나은 경우도 많다.

인덱스가 항상 검색에만 사용되는 것은 아니므로 여러 가지용도를 고려해 적절히 인덱스를 설계할 필요가 있습니다.

**정렬 최적화:** 인덱스는 데이터를 정렬된 상태로 유지합니다. 따라서 인덱스를 사용하면 **ORDER BY** 쿼리에서 디스크 기반의 정렬 작업을 피할 수 있어, 특히 큰 데이터셋에 대한 정렬 작업이 효율적으로 이루어 질 수 있습니다.

**그룹핑 성능 개선:** **GROUP BY** 작업은 정렬된 데이터에 대해 더 빠르게 수행될 수 있습니다. 인덱스를 사용하면 그룹핑 작업에 필요한 전처리 시간이 줄어들며, 결과적으로 쿼리의 전체 성능이 향상될 수 있습니다.

country라는 컬럼과 city라는 컬럼이 포함된 tb\_test 테이블이 있습니다.

tb\_test 테이블의 전체 레코드 건수는 1만 건이며, country 컬럼으로만 인덱스가 생성된 상태에서 아래의 두 케이스를 살펴보겠습니다.

- caseA: country 컬럼의 유니크한 값의 개수가 10개
- caseB: country 컬럼의 유니크한 값의 개수가 1,000개

```
select *
  from tb_test
 where country='KOREA' AND city='SEOUL'
```

MySQL에서는 인덱스의 통계 정보(유니크한 값의 개수)가 관리되기 때문에 city 컬럼의 기수성은 작업 범위에 아무런 영향을 미치지 못한다.(실행계획을 보면 알 수 있음)

위 쿼리를 실행하면 A케이스의 경우 평균 1000건

B케이스의 경우에는 평균 10건이 조회될 수 있다는 것을 인덱스의 통계 정보(유니크한 값의 개수)로 예측할 수 있다.

A케이스와 B케이스 모두 실제 모든 조건을 만족하는 레코드는 단 1건만 있었다면 A 케이스의 인덱스는 적합하지 않은 것이라고 볼 수 있다.

그렇기에 A케이스의 경우 country 컬럼으로 생성된 인덱스는 비효율적입니다.

B케이스도 모든 컬럼이 유니크하지 않지만 현실적으로 조건을 만족하게 생성한다는 것은 불가능하지만 이정도의 낭비는 무시할 수 있습니다.(조회 속도를 위해서)

## 읽어야 하는 레코드의 건수

인덱스를 통해 테이블의 레코드를 읽는 것은 인덱스를 거치지 않고 바로 테이블의 레코드를 읽는 것보다 높은 비용이 드는 작업입니다.

예를 들어 테이블에 레코드가 100만 건이 저장돼 있는데, 그중에서 50만 건을 읽어야 하는 쿼리가 있다고 가정해 보겠습니다.

이 작업은 전체 테이블을 모두 읽어서 필요 없는 50만 건을 버리는 것이 효율적일지, 인덱스를 통해 필요한 50만 건만 읽어 오는 것이 효율적일지 판단해야 합니다.

인덱스를 이용한 손익 분기점이 얼마인지 판단할 필요가 있는데, 일반적인 DBMS의 옵티마이저에서는 인덱스를 통해 레코드를 1건 읽는 것이 테이블에서 직접 레코드 1건을 읽는 것보다 4~5배 정도 비용이 더 많이 드는 작업인 것으로 예측합니다.

즉 인덱스를 통해 읽어야 할 레코드의 건수가 전체 테이블 레코드의 20~25%를 넘어서면 인덱스를 이용하지 않고 테이블을 모두 직접 읽어서 필요한 레코드만 가려내는 필터링 방식으로 처리하는 것이 효율적입니다.

전체 100만 건의 레코드 가운데 50만 건을 읽어야 하는 작업은 인덱스의 손익 분기점인 20~25%보다 훨씬 크기 때문에 MySQL 옵티마이저는 인덱스를 이용하지 않고 직접 테이블을 처음부터 끝까지 읽어서 처리할 것입니다.

이렇게 많은 레코드를 읽을 때는 강제로 인덱스를 사용하도록 힌트를 추가해도 성능상 얻을 수 있는 이점이 없습니다.

이러한 작업은 MySQL의 옵티마이저가 기본적으로 힌트를 무시하고 테이블을 직접 읽는 방식으로 처리하겠지만 알아두면 좋습니다.

## B-Tree 인덱스를 통한 데이터 읽기

어떤 경우에 인덱스를 사용하게 유도할지, 혹은 사용하지 못하게 할지 판단하려면 MySQL 스토리지 엔진이 어떻게 인덱스를 이용해서 실제 레코드를 읽어 내는지 알아야 합니다.  
MySQL이 인덱스를 이용하는 대표적인 방법 세 가지를 확인해보겠습니다.

## 인덱스 레인지 스캔

인덱스 레인지 스캔은 인덱스의 접근 방법 가운데 가장 대표적인 접근 방식으로 이후 설명할 두 접근 방식보다 빠른 방법입니다.

보통 인덱스를 통해 레코드를 한 건만 읽는 경우와 한 건 이상을 읽는 경우를 각각 다른 이름으로 구분하지만, 책에서는 모두 묶어서 "인덱스 레인지 스캔"이라고 표현합니다.

```
select * from employees where first_name BETWEEN 'Ebbe' And 'Gad'
```

19  
28 ✓ explain select \* from employees where first\_name BETWEEN 'Ebbe' AND 'Gad'

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	<null>	range	ix_firstname	ix_firstname	58	<null>	27714	100	Using index condition

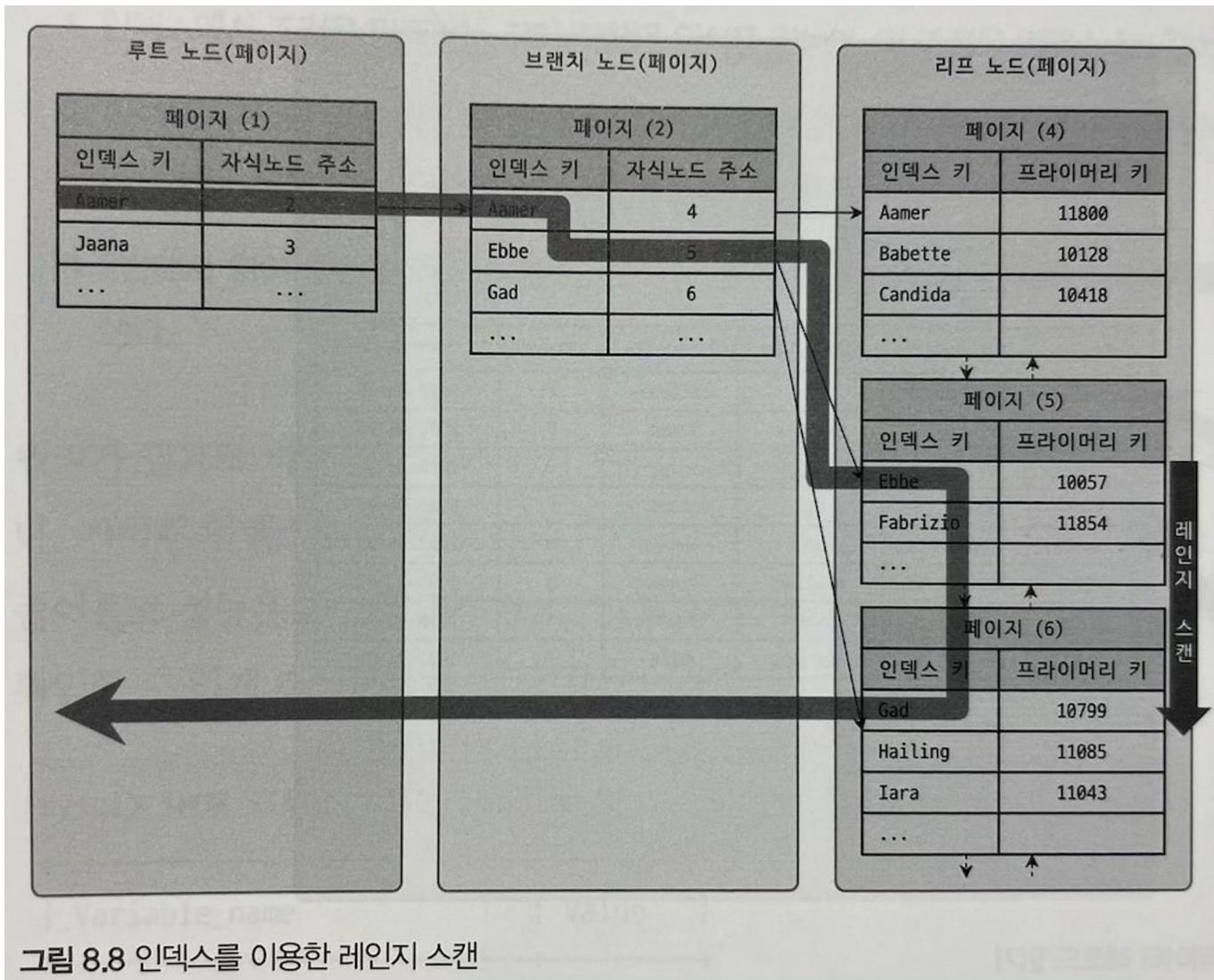


그림 8.8 인덱스를 이용한 레인지 스캔

인덱스 레인지 스캔은 검색해야 할 인덱스의 범위가 결정됐을 때 사용하는 방식입니다.

일반적으로 검색하려는 값의 수나 검색 결과 레코드 건수와 관계없이 레인지 스캔이라고 표현합니다.

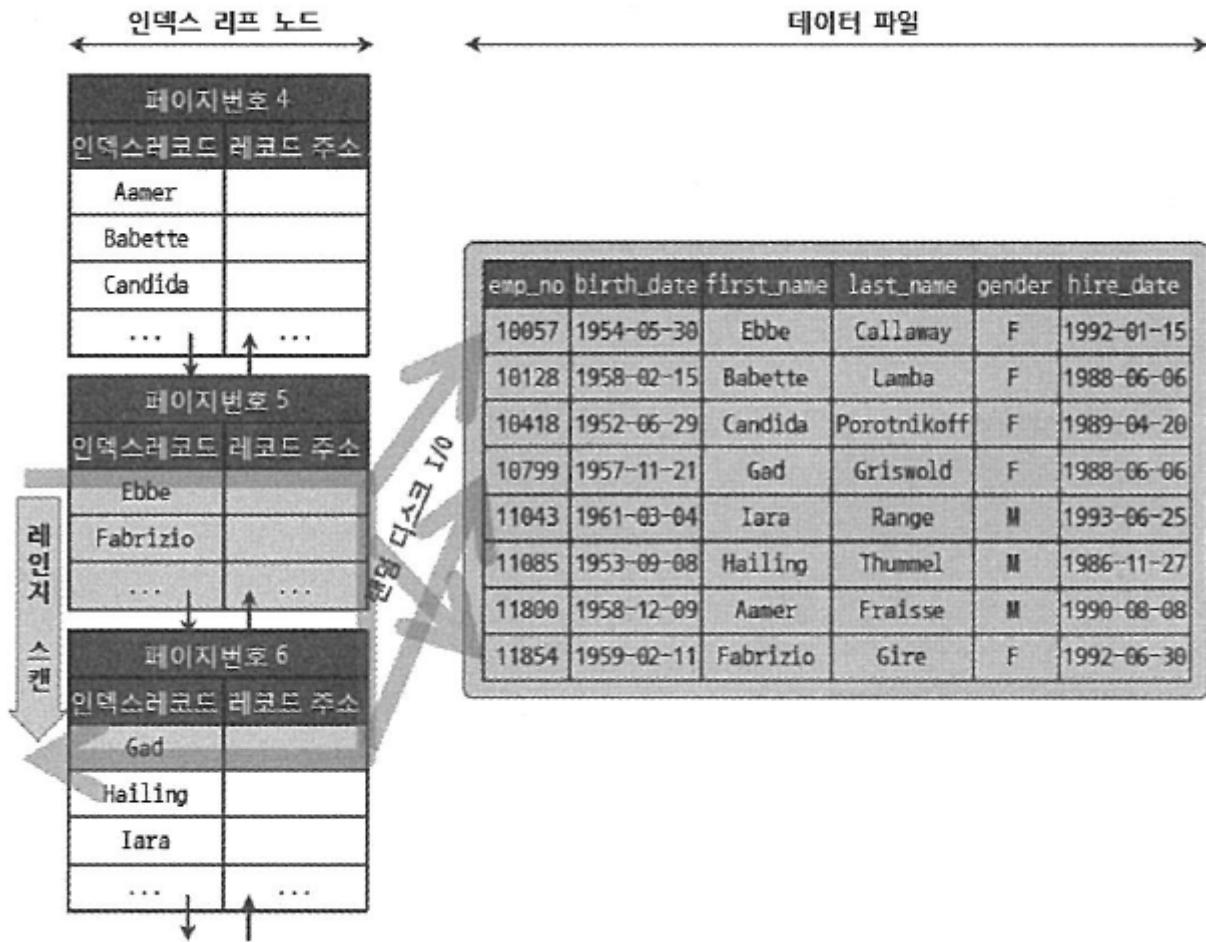
위 그림의 화살표에서 알 수 있듯이

1. 루트 노드에서부터 비교를 시작해 브랜치 노드를 거치고 최종적으로 리프 노드까지 찾아서 들어가면서 레코드의 시작 지점을 찾습니다.
2. 시작 위치를 찾으면 그 이후부터는 리프노드의 레코드만 순서대로 읽습니다.  
이처럼 차례대로 쭉 읽는 것을 스캔이라고 표현합니다.
3. 만약 스캔하다가 리프 노드의 끝까지 읽으면 리프 노드간의 링크를 이용해 다음 리프 노드를 찾아서 다시 스캔합니다.

4. 최종적으로 스캔을 멈춰야 할 위치에 다다르면 지금까지 읽은 레코드를 사용자에게 반환하고 쿼리를 끝냅니다.

위 그림에서 두꺼운 선을 스캔해야 할 위치 검색을 위한 비교 작업을 의미하며, 두꺼운 화살표가 지나가는 리프 노드의 레코드 구간은 실제 스캔하는 범위를 표현합니다

아래 그림은 B-Tree 인덱스의 리프 노드를 스캔하면서 실제 데이터 파일의 레코드를 읽어오는 과정입니다.



[그림 5-10] 인덱스 레인지 스캔을 통한 데이터 레코드 읽기

위 그림은 B-Tree 인덱스에서 루트와 브랜치 노드를 이용해 스캔 시작 위치를 검색하고, 해당 지점부터 필요한 방향(ASC,DESC)로 인덱스를 읽어 나가는 과정입니다.

여기서 살펴봐야 할 것은 어떤 방식으로 스캔하든 관계없이, 해당 인덱스를 구성하는 컬럼의 정순 또는 역순으로 정렬된 상태로 레코드를 가져옵니다.

이는 B-Tree의 특징으로 데이터를 삽입하는 순간부터 정렬되기 때문입니다.

또한 인덱스의 리프 노드에서 검색 조건에 일치하는 건들은 데이터 파일에서 레코드를 읽어오는 과정이 필요합니다. 이때마다 리프노드에서 저장된 레코드 주소로 데이터 파일의 레코드를 읽어오는데, 레코드 한 건 한 건 마다 랜덤 IO가 한 번씩 발생합니다.

만약 3건의 레코드가 검색 조건에 일치했다고 가정하면, 데이터 레코드를 읽기 위해 랜덤 IO가 최대 3번 발생

합니다.

그래서 인덱스를 통해 데이터 레코드를 읽는 작업은 비용이 많이 드는 작업이라고 할 수 있습니다.

앞서 이야기 했듯이 데이터 레코드가 20~25%를 넘으면 인덱스를 통한 읽기보다는 테이블의 데이터를 직접 읽는 것이 효율적인 처리 방식인 것도 납득이 갑니다(인덱스 비용 4, 그냥 읽기 1)

정리 하면 인덱스 레인지 스캔은 다음과 같이 크게 3단계를 거칩니다.

1. 인덱스 탐색(인덱스에서 조건을 만족하는 값이 저장된 위치 찾기)
2. 인덱스 스캔(탐색된 위치부터 필요한 만큼 인덱스를 차례대로 쭉 읽는다. 인덱스 탐색과정도 인덱스 스캔이라고 할 수 있음)
3. 인덱스 스캔 과정에서 읽어 들인 인덱스 키와 레코드 주소를 이용해 레코드가 저장된 페이지를 가져오고, 최종 레코드를 읽어온다

### 커버링 인덱스

- 쿼리가 필요로 하는 데이터에 따라 3번 과정은 필요하지 않을 수도 있는데, 이를 커버링 인덱스라고 합니다.
- 커버링 인덱스로 처리되는 쿼리는 디스크의 레코드를 읽지 않아도 되기 때문에 랜덤 읽기가 상당히 줄어들고, 성능은 그만큼 빨라집니다.
- MySQL 서버에서는 1번과 2번 단계의 작업이 얼마나 수행됐는지를 확인할 수 있게 다음과 같은 상태 값을 제공합니다.

```
mysql> show status like 'Handler_read_%';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| Handler_read_first | 17      |
| Handler_read_key   | 986     |
| Handler_read_last  | 0       |
| Handler_read_next  | 679     |
| Handler_read_prev  | 0       |
| Handler_read_rnd   | 0       |
| Handler_read_rnd_next | 300068 |
+-----+-----+
7 rows in set (0.01 sec)
```

먼저 새 커넥션을 연결한 직후의 상태 값입니다.

```
1 of 2 | 4
2
3
4
5 ✓ SHOW STATUS LIKE 'Handler_read_%'
```

Output Result 2 Plan

Variable_name	Value
Handler_read_first	0
Handler_read_key	0
Handler_read_last	0
Handler_read_next	0
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_next	665

DataGrip에서 500건의 데이터만 제한하기 때문에 Handler\_read\_next의 값은 500입니다.

여기서 Handler\_read\_next는 인덱스 정순으로 읽은 레코드 건수입니다.

```
1 of 2 | 3
2
3 select * from employees where first_name BETWEEN 'Ebbe' AND 'Gad';
4
5 ✓ SHOW STATUS LIKE 'Handler_read_%'
```

Output Result 6 Plan

Variable_name	Value
Handler_read_first	0
Handler_read_key	1
Handler_read_last	0
Handler_read_next	500
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_next	665

여기서 first\_name 기준으로 역순으로 읽은 경우 Handler\_read\_prev의 값은 500이 됩니다.

참고로 Handler\_read\_prev는 인덱스 역순으로 읽은 레코드 건수입니다.

```

3 select * from employees where first_name BETWEEN 'Ebbe' AND 'Gad';
4 select * from employees where first_name BETWEEN 'Ebbe' AND 'Gad' order by first_name desc;
5 ✓ SHOW STATUS LIKE 'Handler_read_%'

```

**Output** Result 8 × Plan ×

Variable_name	Value
Handler_read_first	0
Handler_read_key	2
Handler_read_last	0
Handler_read_next	500
Handler_read_prev	500
Handler_read_rnd	0
Handler_read_rnd_next	665

여기서 min, max를 각각 호출하는 경우 Handler\_read\_first 와 Handler\_read\_last의 값이 각각 1씩 증가한 것을 확인할 수 있습니다.

```

5 select min(employees.emp_no) from employees;
6 select max(employees.emp_no) from employees;
7 ✓ SHOW STATUS LIKE 'Handler_read_%'

```

**Output** Result 18 × Plan ×

Variable_name	Value
Handler_read_first	1
Handler_read_key	6
Handler_read_last	1
Handler_read_next	15743
Handler_read_prev	500
Handler_read_rnd	0
Handler_read_rnd_next	665

그 외

- Handler\_read\_rnd : 고정된 위치를 근거로 열을 읽기 위한 요청 횟수. 결과 값을 정렬하도록 요청하는 많은 수의 쿼리를 실행하는 중이라면, 이 값이 높게 설정된다. 이것은 MySQL이 전체 테이블을 스캔할 것을 요구하는 쿼리를 많이 가지고 있거나 또는 키를 정확히 사용하지 않는 조인(join)을 가지고 있는 것이다.

- Handler\_read\_rnd\_next : 데이터 파일에 있는 다음 열을 읽기 위한 요청 횟수. 이 값은 여러분이 많은 수의 테이블 스캔을 할 경우에 높게 된다. 이것은 일반적으로 테이블이 올바르게 인덱스 되지 않았거나 또는 쿼리가 인덱스를 제대로 활용하지 못하고 있음을 의미한다.

## 인덱스 풀 스캔

인덱스 풀 스캔은 인덱스 레인지 스캔과 달리 처음부터 끝까지 모두 읽는 방식입니다.

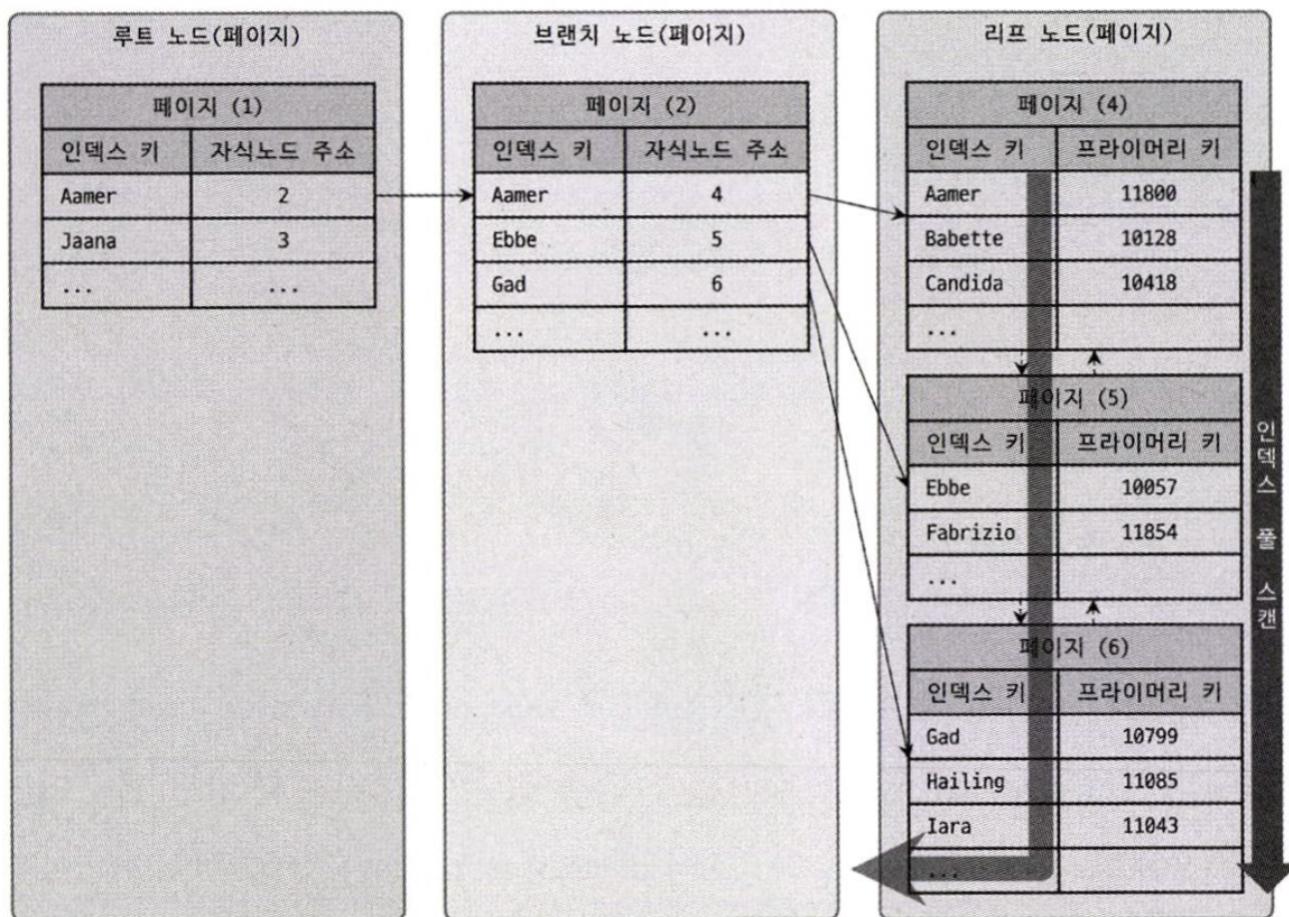
대표적으로 쿼리의 조건절에 사용된 컬럼이 인덱스의 첫 번째 컬럼이 아닌 경우 인덱스 풀 스캔 방식이 사용됩니다.

예를 들어, 인덱스 (A,B,C)컬럼의 순서로 만들어져 있을 때 조건절에 B 혹은 C를 선택하는 경우에 해당합니다.

일반적으로 인덱스의 크기는 테이블의 크기보다 작기 때문에 직접 테이블을 처음부터 끝까지 읽는 것보다는 인덱스만 읽는 것이 효율적입니다.

쿼리가 인덱스에 명시도니 컬럼만으로 조건을 처리하는 경우 이 방식을 사용하는 것을 권장합니다.

마찬가지로 인덱스는 레코드의 20~25%를 읽는 경우 이 방식을 사용하지 않습니다.



위 그림은 인덱스 풀 스캔의 예시로 인덱스 리프 노드의 제일 앞 또는 제일 뒤로 이동한 후, 인덱스의 리프 노드를 연결하는 링크드 리스트를 따라서 처음부터 끝까지 스캔하는 방식입니다.

단 이 방식은 인덱스 레인지 스캔보다는 빠르지 않지만 테이블 풀 스캔보다는 효율적입니다.

(이해가 가지 않지만) 앞서 언급한 예시 (A,B,C)의 경우 인덱스에 포함된 컬럼만으로 쿼리를 처리할 수 있는

경우 테이블의 레코드를 읽을 필요가 없기 때문에 인덱스 테이블의 전체 크기는 테이블 자체의 크기보다는 훨씬 작기 때문에 인덱스 풀 스캔은 테이블 전체를 읽는 것보다는 적은 디스크 IO로 처리할 수 있습니다.

#### ▲ 주의

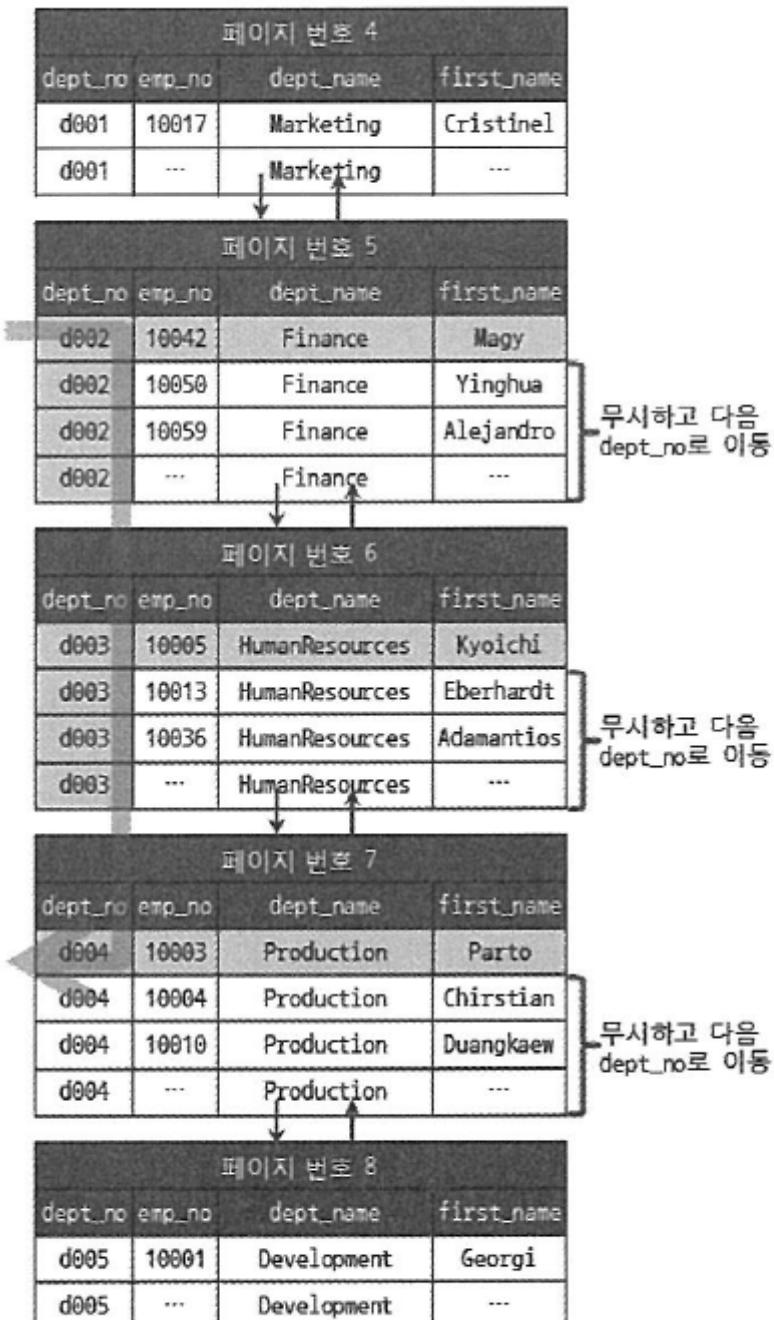
책에서 특별히 방식을 언급하지 않고 인덱스를 사용한다고 표현하면 인덱스 레인지 스캔 혹은 루스 인덱스 스캔방식으로 인덱스를 사용하는 것을 의미합니다.

인덱스 풀 스캔 방식 또한 인덱스를 이용하는 것이지만 효율적인 방식은 아니며, 일반적으로 인덱스를 생성하는 것은 목적이 아닙니다. 반대로 테이블 전체를 읽거나 인덱스 풀 스캔 방식으로 인덱스를 사용하는 경우는 인덱스를 사용하지 못한다 혹은 효율적으로 사용하지 못한다에 가깝다고 이해하면 좋습니다.

## 루스 인덱스 스캔

- 루스 인덱스 스캔은 오라클과 같은 DBMS의 인덱스 스kip 스캔이라고 하는 기능과 작동 방식은 비슷합니다.
- MySQL 5.7 버전까지는 MySQL의 루스 인덱스 스캔 기능이 많이 제한적이었지만 8.0버전 부터는 다른 사용 DBMS에서 지원하는 인덱스 스kip 스캔과 같은 최적화를 조금씩 지원하기 시작했습니다.
- 앞에서 소개한 인덱스 레인지 스캔과 인덱스 풀 스캔은 루스 인덱스 스캔과는 상반된 의미에서 타이트 인덱스 스캔으로 분류 합니다.

- 루스 인덱스 스캔이란 말 그대로 느슨하게 혹은 흔성흔성하기 인덱스를 읽는 것을 의미합니다.



[그림 5-12] 루스 인덱스 스캔(dept\_name과 first\_name 컬럼은 참조용으로 표시됨)

(루스 인덱스 스캔(dept\_name과 first\_name 컬럼은 참조용으로 표시))

- 루스 인덱스 스캔은 인덱스 레인지 스캔과 비슷하게 작동하지만 중간에 필요치 않은 인덱스 키 값은 무시하고 다음으로 넘어가는 형태로 처리합니다.
- 일반적으로 GROUP BY 또는 집합 함수 가운데 MAX 또는 MIN 함수에 대해 최적화를 하는 경우에 사용됩니다.

```

select dept_no, min(emp_no) from dept_emp
where dept_no between 'd002' and 'd004'
group by dept_no;

```

extra에서 Using index for group-by라고 표시 되어 있으면 루스 인덱스 스캔을 사용한 것임

```

+-----+-----+
| id  | select_type |
+-----+-----+
| 1   | SIMPLE      |
+-----+-----+
| id  | table       |
+-----+-----+
| 1   | dept_emp    |
+-----+-----+
| id  | partitions |
+-----+-----+
| null| <null>     |
+-----+-----+
| id  | type        |
+-----+-----+
| 2   | range       |
+-----+-----+
| id  | possible_keys |
+-----+-----+
| 1,2 | PRIMARY,ix_fromdate,ix_empno_fromdate |
+-----+-----+
| id  | key          |
+-----+-----+
| 1   | PRIMARY      |
+-----+-----+
| id  | key_len      |
+-----+-----+
| 16  | 16           |
+-----+-----+
| id  | ref          |
+-----+-----+
| null| <null>     |
+-----+-----+
| id  | rows         |
+-----+-----+
| 4   | 4            |
+-----+-----+
| id  | filtered     |
+-----+-----+
| 100 | 100          |
+-----+-----+
| id  | Extra        |
+-----+-----+
| 1   | Using where; Using index for group-by |
+-----+-----+

```

- 위 쿼리에서 사용된 dept\_emp 테이블은 dept\_no와 emp\_no라는 두 개의 컬럼으로 인덱스가 생성돼 있다.
- 또한 이 인덱스는 (dept\_no, emp\_no) 조합으로 정렬까지 돼 있어서 위 그림에서 같이 dept\_no 그룹 별로 첫 번째 레코드의 emp\_no 값만 읽으면 된다.
- 즉 인덱스에서 where 조건을 만족하는 범위 전체를 다 스캔할 필요가 없다는 것을 옵티마이저는 알고 있기 때문에 조건에 만족하지 않는 레코드는 무시하고 다음 레코드로 이동합니다.
- 위 그림을 보면 인덱스 리프 노드를 스캔하면서 불필요한 부분은 그냥 무시하고 필요한 부분만 읽었음을 알 수 있습니다.
- 루스 인덱스 스캔을 사용하려면 여러 조건을 만족해야 하는데 이후 실행 계획에서 다룰 예정입니다.

여담으로 저런 텍스트가 얼마나 있을지 궁금해서 찾아보니깐 생각보다 많네요?

근데 왜 LooseScan이 아니라 Using index of group by가 루스 스캔인지 잘 모르겠습니다

```

THIS ARRAY MUST BE IN SYNC WITH EXTRA_TAG_ENUM.

static const char *traditional_extra_tags[ET_total] = {
    nullptr,                                // ET_NONE
    "Using temporary",                      // ET_USING_TEMPORARY
    "Using filesort",                       // ET_USING_FILESORT
    "Using index condition",                // ET_USING_INDEX_CONDITION
    "Using",                                 // ET_USING
    "Range checked for each record",        // ET_RANGE_CHECKED_FOR_EACH_RECORD
    "Using pushed condition",               // ET_USING_PUSHED_CONDITION
    "Using where",                          // ET_USING_WHERE
    "Not exists",                           // ET_NOT_EXISTS
    "Using MRR",                            // ET_USING_MRR
    "Using index",                          // ET_USING_INDEX
    "Full scan on NULL key",               // ET_FULL_SCAN_ON_NULL_KEY
    "Using index for group-by",            // ET_USING_INDEX_FOR_GROUP_BY
    "Using index for skip scan",           // ET_USING_INDEX_FOR_SKIP_SCAN,
    "Distinct",                             // ET_DISTINCT
    "LooseScan",                            // ET_LOOSESCAN
    "Start temporary",                     // ET_START_TEMPORARY
    "End temporary",                       // ET_END_TEMPORARY
    "FirstMatch",                           // ET_FIRST_MATCH
    "Materialize",                          // ET_MATERIALIZE
    "Start materialize",                   // ET_START_MATERIALIZE
    "End materialize",                     // ET_END_MATERIALIZE
    "Scan",                                 // ET_SCAN
    "Using join buffer",                  // ET_USING_JOIN_BUFFER
    "const row not found",                // ET_CONST_ROW_NOT_FOUND
    "unique row not found",               // ET_UNIQUE_ROW_NOT_FOUND
    "Impossible ON condition",            // ET_IMPOSSIBLE_ON_CONDITION
    "",                                    // ET_PUSHED_JOIN
    "Ft_hints:",                           // ET_FT_HINTS
    "Backward index scan",                // ET_BACKWARD_SCAN
    "Recursive",                            // ET_RECURSIVE
    "Table function:",                    // ET_TABLE_FUNCTION
    "Index dive skipped due to FORCE",   // ET_SKIP_RECORDS_IN_RANGE
    "Using secondary engine",              // ET_USING_SECONDARY_ENGINE
    "Rematerialize"                       // ET_REMATERIALIZE
};


```

## 인덱스 스kip 스캔

- 데이터베이스 서버에서 인덱스의 핵심은 값이 정렬돼 있다는 것이며, 이로 인해 인덱스를 구성하는 컬럼의 순서가 매우 중요합니다.
- 예를 들어 employees 테이블에 다음과 같이 인덱스를 생성해봅니다.

```
alter table employees ADD index ix_gender_birthdate (gender, birth_date);
```

- 위 인덱스를 사용하려면 where 조건절에 gender 컬럼에 대한 비교 조건이 필수입니다.

```
# 인덱스를 사용하지 못하는 쿼리
```

```
SELECT * FROM employees where birth_date >= '1965-02-01';
```

```
# 인덱스를 사용할 수 있는 쿼리
```

```
SELECT * FROM employees where gender='M' AND birth_date >= '1965-02-01';
```

- 위 두 쿼리 중에서 gender 컬럼과 birth\_date 컬럼의 조건을 모두 가진 두 번째 쿼리는 인덱스를 효율적으로 사용할 수 있지만 gender 컬럼에 대한 비교 조건이 없는 첫 번째 쿼리는 인덱스를 사용할 수가 없었다.
- 주로 이런 경우에는 birth\_date 컬럼부터 시작하는 인덱스를 새로 생성해야만 했다.
- MySQL 8.0 버전부터는 옵티마이저가 gender 컬럼을 건너뛰어서 birth\_date 컬럼만으로도 인덱스 검색이 가능하게 해주는 인덱스 스kip 스캔 (Index skip scan) 최적화 기능이 도입됐다.
- 물론 MySQL 8.0 이전 버전에서도 인덱스 스kip 스캔과 비슷한 최적화를 수행하는 루스 인덱스 스캔이라는 기능이 있었지만 루스 인덱스 스캔은 GROUP BY 작업을 처리하기 위해 인덱스를 사용하는 경우에만 적용할 수 있었다.
- 하지만 MySQL 8.0 버전에 도입된 스kip 스캔은 WHERE 조건절의 검색을 위해 사용 가능하도록 용도가 훨씬 넓어진 것이다.
- 우선 인덱스 스kip 스캔 기능을 비활성화하고, MySQL 8.0 이전 버전에서 어떻게 실행 계획으로 처리됐는지 한 번 살펴보자.

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:

```
set optimizer_switch = 'skip_scan=off';
explain select gender, birth_date from employees where birth_date >= '1965-02-01'
```
- Result Grid:

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1 SIMPLE	employees	<null>	index	<null>	ix_gender_birthdate	4	<null>	299772	33.33	Using where; Using index

- 위 쿼리는 WHERE 조건절에 gender 컬럼에 대한 조건 없이 birth\_date 컬럼의 비교 조건만 가지고 있기 때문에 쉽게 ix\_gender\_birthdate 인덱스를 효율적으로 이용할 수 없다.

- 여기서 인덱스를 효율적으로 이용한다는 것은 일반적으로 우리가 인덱스를 이용한다라는 표현과 동일한 의미로, 인덱스에서 꼭 필요한 부분만 접근하는 것을 의미합니다.
- 위의 실행 계획에서 type 컬럼이 "index"라고 표시된 것은 인덱스를 처음부터 끝까지 모두 읽었다(풀 인덱스 스캔)는 의미이므로 인덱스를 비효율적으로 사용한 것이다.
- 위 예제 쿼리는 인덱스에 있는 gender 컬럼과 birth\_date 컬럼만 있으면 처리를 완료할 수 있기 때문에 ix\_gender\_birthdate 인덱스를 풀 스캔한 것이다.
- 만약 예제쿼리가 employees 테이블의 모든 컬럼을 가져와야 했다면 테이블 풀 스캔을 실행했을 것이다.
- MySQL 8.0 버전부터 도입된 인덱스 스킵 스캔을 활성화하고, 동일 쿼리의 실행 계획을 다시 확인해보자



```
explain select gender, birth_date from employees where birth_date >= '1965-02-01'
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	<null>	range	ix_gender_birthdate	ix_gender_birthdate	4	<null>	99914	100	Using where; Using index for skip scan

- 이번에는 쿼리의 실행 계획에서 type 컬럼의 값이 "range"로 표시됐는데, 이는 인덱스에서 꼭 필요한 부분만 읽었다는 것을 의미한다
- 그리고 실행 계획의 Extra 컬럼에 "Using index of skip scan"이라는 문구가 표시됐는데, 이는 ix\_gender\_birthdate 인덱스에 대해 인덱스 스킵 스캔을 활용해 데이터를 조회했다는 것을 의미합니다.
- MySQL 옵티マイ저는 우선 gender 컬럼에서 유니크한 값을 모두 조회해서 주어진 쿼리에 gender 컬럼의 조건을 추가해서 쿼리를 다시 실행하는 형태로 처리한다.

아래 그림은 인덱스 스kip 스캔을 어떻게 처리되는지를 보여준다.

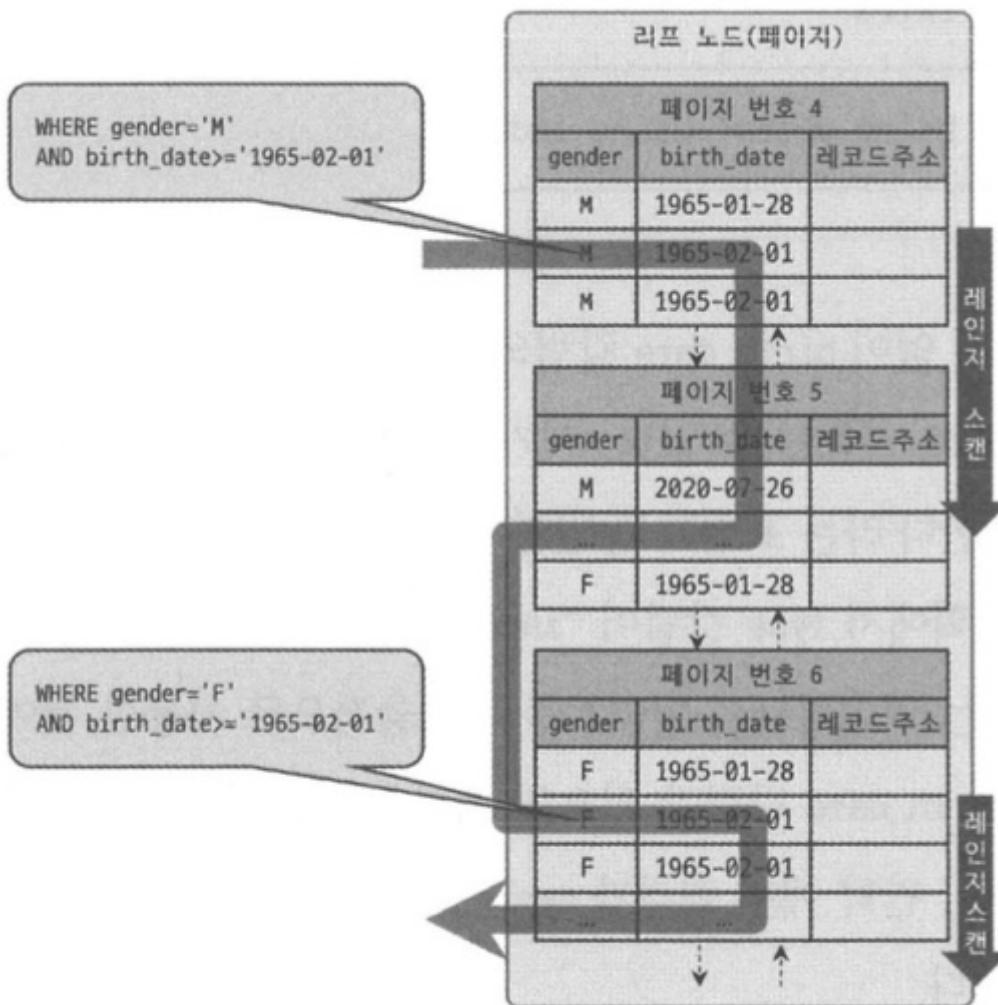


그림 8.12 인덱스 스kip 스캔

- gender 컬럼은 성별을 구분하는 컬럼으로 'M'과 'F'값만 가지는 Enum 타입의 컬럼입니다.
- 그래서 gender 컬럼에 대해 가능한 2개 ('M'과 'F')를 구한 다음, 옵티마이저는 내부적으로 아래 2개의 쿼리를 실행하는 것과 비슷한 형태의 최적화를 실행하게 된다.

```
select gender, birth_date from employees where gender='F' AND birth_date >= '1965-02-01'  
select gender, birth_date from employees where gender='M' AND birth_date >= '1965-02-01'
```

#### △ 주의

여기서 gender 컬럼이 enum('M', 'F') 타입이기 때문에 이런 처리가 가능한 것은 아닙니다.

컬럼이 어떤 값을 가지더라도 MySQL 서버는 인덱스를 루스 인덱스 스캔과 동일한 방식으로 읽으면서 인덱스에 존재하는 모든 값을 먼저 추출하고 그 결과를 이용해 인덱스 스kip 스캔을 실행합니다.

- 인덱스 스kip 스캔은 MySQL 8.0 버전에 새롭게 도입된 기능이어서 아직 다음과 같은 단점이 있습니다.
  - Where 조건절에 조건이 없는 인덱스의 선행 컬럼의 유니크한 값의 개수가 적어야 함
  - 쿼리가 인덱스에 존재하는 컬럼만으로 처리 가능해야 함 (커버링 인덱스)

- 첫 번째 조건은 쿼리 실행 계획의 비용과 관련된 부분인데, 만약 유니크한 값의 개수가 매우 많다면 MySQL 옵티마이저는 인덱스에서 스캔해야 할 시작 지점을 검색하는 작업이 많이 필요해진다.
- 그래서 쿼리의 처리 성능이 오히려 더 느려질 수도 있다.
- 예를 들어 (emp\_no, dept\_no) 조합으로 만들어진 인덱스에서 스kip 스캔을 실행한다고 가정하면 사원의 수만큼 레인지 스캔 시작 지점을 검색하는 작업이 필요해져 쿼리의 성능이 매우 떨어진다.
- 그래서 인덱스 스kip 스캔은 인덱스의 선행 컬럼이 가진 유니크한 값의 개수가 소량일 때만 적용 가능한 최적화라는 점을 기억해야 한다.

```
explain select * from employees where birth_date >= '1965-02-01';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	<null>	ALL	<null>	<null>	<null>	<null>	299772	33.33	Using where

- 위의 쿼리는 WHERE 조건절은 동일하지만 select 절에서 employees 테이블의 모든 컬럼을 조회하도록 변경했다.
- 이 쿼리는 ix\_gender\_birthdate 인덱스에 포함된 gender 컬럼과 birth\_date 컬럼 이외의 나머지 컬럼도 필요로 하기 때문에 인덱스 스kip 스캔을 사용하지 못하고 풀 테이블 스캔으로 실행 계획을 수립한 것을 확인할 수도 있다.
- 하지만 이 제약 사항은 MySQL 서버의 옵티마이저가 개선되면 충분히 해결될 수 있는 부분이다.

## 다중 컬럼(Multi-column) 인덱스

이전까지 살펴본 인덱스들은 모두 1개의 컬럼만 포함된 인덱스입니다. 보통 실제 서비스에서는 2개 이상의 컬럼을 포함하는 인덱스가 더 많이 사용하는데

이를 다중컬럼 인덱스 또는 복합 컬럼 인덱스, Concatenated Index(두 개 이상의 컬럼이 연결)라고 합니다.

아래 그림에서 2개 이상의 컬럼을 포함하는 다중 컬럼 인덱스의 구조 예시입니다.

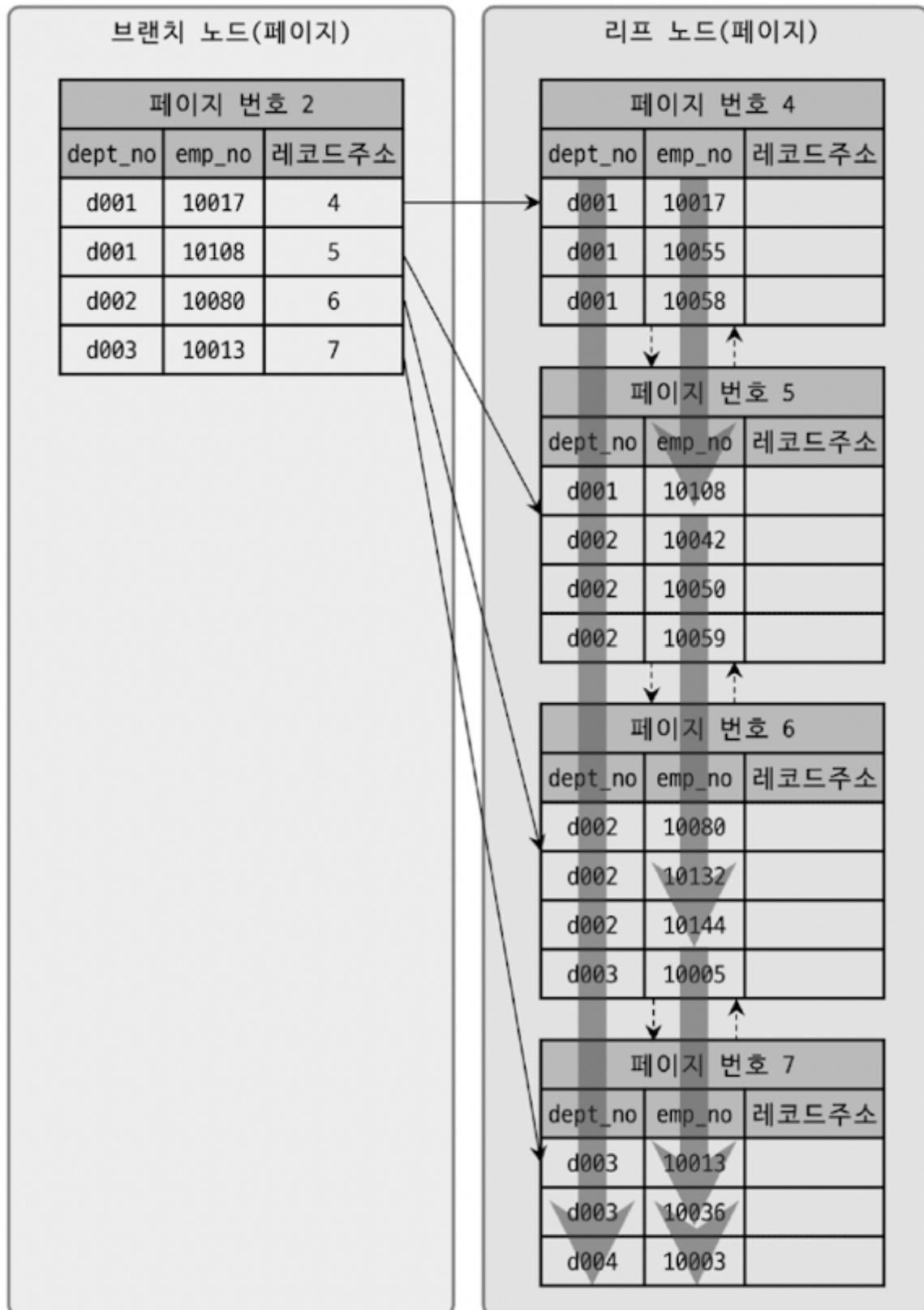


그림 8.13 다중 칼럼 인덱스

위 그림에서 루트 노드가 존재하지 않는데 이는 생략된 것이고 실제로는 데이터 레코드 건수가 작은 경우에는 브랜치 노드가 없는 경우는 있습니다.

참고로 위 그림에서 다중 컬럼 인덱스일 때 인덱스를 구성하는 컬럼의 값이 어떻게 정렬되어 있는지를 보여줍니다.

눈여겨 봐야할 점은 위 그림에서 두 번째 컬럼은 첫 번째 컬럼에 의존해서 정렬된다는 점입니다. (첫 번째 컬럼 순서로 정렬된 이후 두 번째 컬럼으로 정렬됨)

그래서 다중 컬럼 인덱스에서는 인덱스 내에서 각 컬럼의 위치(순서)가 상당히 중요하며, 신중히 결정해야 합니다.

## B-Tree 인덱스의 정렬 및 스캔 방향

- 인덱스를 생성할 때 설정한 정렬 규칙에 따라서 인덱스의 키 값은 항상 오름차순이거나 내림차순으로 정렬되어 저장됩니다.
- 하지만 어떤 인덱스가 오름차순으로 생성됐다고 해서 그 인덱스를 오름차순으로만 읽을 수 있다는 뜻은 아닙니다.
- 사실 그 인덱스를 거꾸로 끝에서부터 읽으면 내림차수로 정렬된 인덱스로도 사용될 수 있습니다.
- 인덱스를 어느 방향으로 읽을지는 쿼리에 따라 옵티마이저가 실시간으로 만들어 내는 실행 계획에 따라 결정됩니다.

## 인덱스의 정렬

- 일반적인 상용 DBMS에서는 인덱스를 생성하는 시점에 인덱스를 구성하는 각 컬럼의 정렬을 오름차순 또는 내림차순으로 설정할 수 있다.
- MySQL 5.7 버전까지는 컬럼 단위로 정렬 순서를 혼합해서 인덱스를 생성할 수 없었다.
- 이런 문제점을 해결하기 위해 숫자 컬럼의 경우 -1을 곱한 값을 저장하는 우회 방법을 사용했었다.
- 현재는 다음과 같은 형태의 정렬 순서를 혼합한 인덱스도 생성할 수 있게 됐다

```
| CREATE index ix_teamname_userscore on employees ( team_name asc, user_score desc);
```

### △ 주의

아마도 MySQL 5.7에서도 위와 같이 오름차순 컬럼과 내림차순 컬럼을 혼합한 인덱스를 생성했다고 기억하는 사용자도 있을 것입니다.

실제 이렇게 인덱스를 생성해도 아무런 에러 없이 인덱스가 생성됐을 것이다.

하지만 실제 인덱스는 모두 오름차순 정렬만으로 인덱스가 생성됐다.

MySQL 5.7버전까지는 ASC 또는 DESC 키워드는 앞으로 만들어질 버전에 대한 호환성을 위해 문법상으로만 제공된 것이다.

## 인덱스 스캔 방향

first\_name을 기준으로 역순으로 조회하는 쿼리를 예시로 보겠습니다.

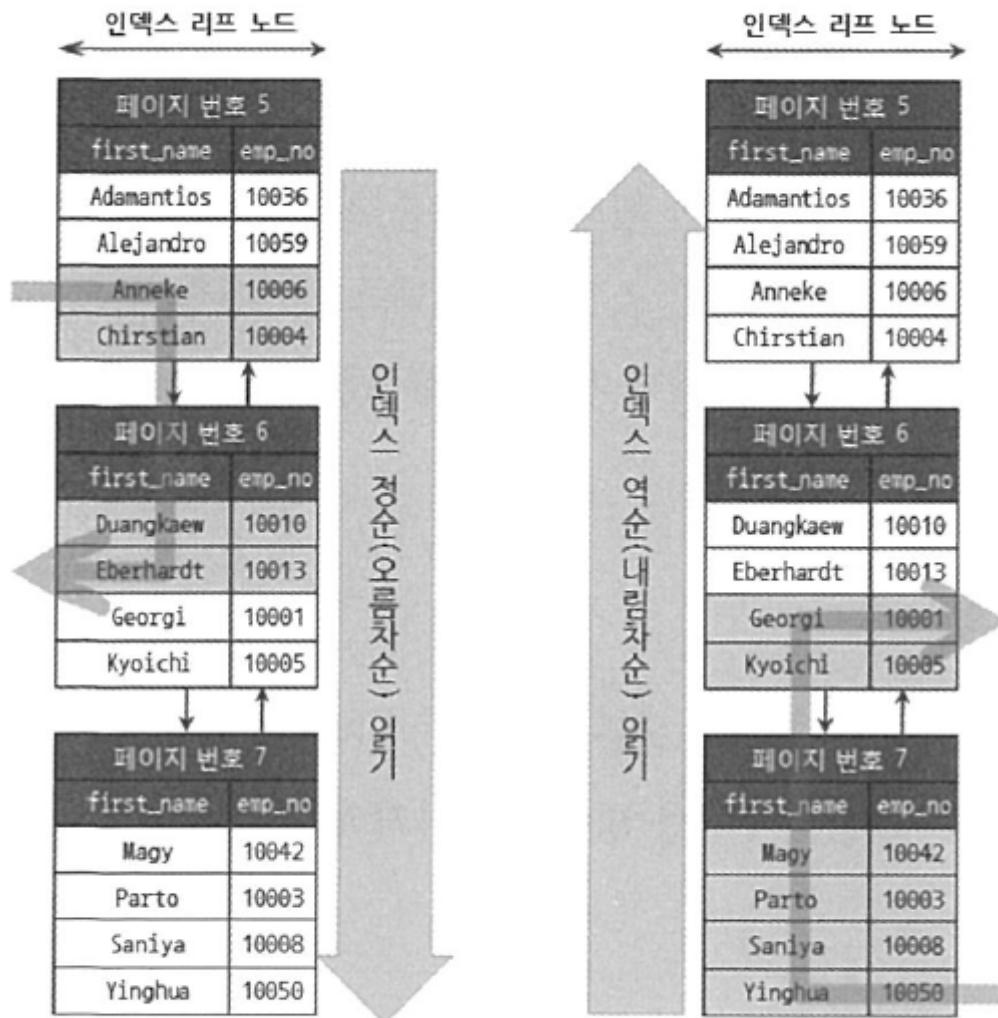
아래 쿼리를 봤을 때 인덱스를 처음부터 오름차순으로 끝까지 읽어 first\_name이 가장 큰 값 하나를 가져온다고 유추할 수 있지만 실제로는 그렇지 않습니다.

옵티마이저의 경우 인덱스를 최솟값부터 읽으면 오름차순으로 값을 가져오고 최대값부터 거꾸로 읽으면 내림

차순으로 값을 가져올 수 있다는 것을 알고 있습니다.

```
SELECT * FROM employees order by first_name desc limit 1;
```

- 아래 그림은 인덱스를 정순으로 읽는 경우와 역순으로 읽는 경우를 보여줍니다.



[그림 5-14] 인덱스의 오름차순(ASC)과 내림차순(DESC) 읽기

```
SELECT * FROM employees where first_name >= 'Anneke'  
order by first_name ASC limit 4;
```

```
select * from employees order by first_name desc limit 5;
```

인덱스 생성 시점에 오름차순 혹은 내림차순으로 정렬이 결정되지만 쿼리가 그 인덱스를 사용하는 시점에 인덱스를 읽는 방향에 따라 오름차순 또는 내림차순 정렬 효과를 얻을 수 있습니다.

오름차순으로 생성된 인덱스를 정순으로 읽으면 출력되는 결과 레코드는 자동으로 오름차순으로 정렬된 결과가 되고, 역순으로 읽으면 그 결과는 내림차순으로 정렬된 상태입니다.

위 코드를 보면 첫 번째 쿼리는 first\_name 컬럼에 정의된 인덱스를 이용해 'Anneke'라는 레코드를 찾은 후, 정순으로 해당 인덱스를 읽으면서 4개의 레코드만 가져오면 인덱스를 만드는 비용 외 들이지 않고 정렬 효과를 얻을 수 있습니다.

두 번째 쿼리는 first\_name 컬럼에 정의된 인덱스를 역순으로 읽으면서 처음 다섯 개의 레코드만 가져오면 됩니다.

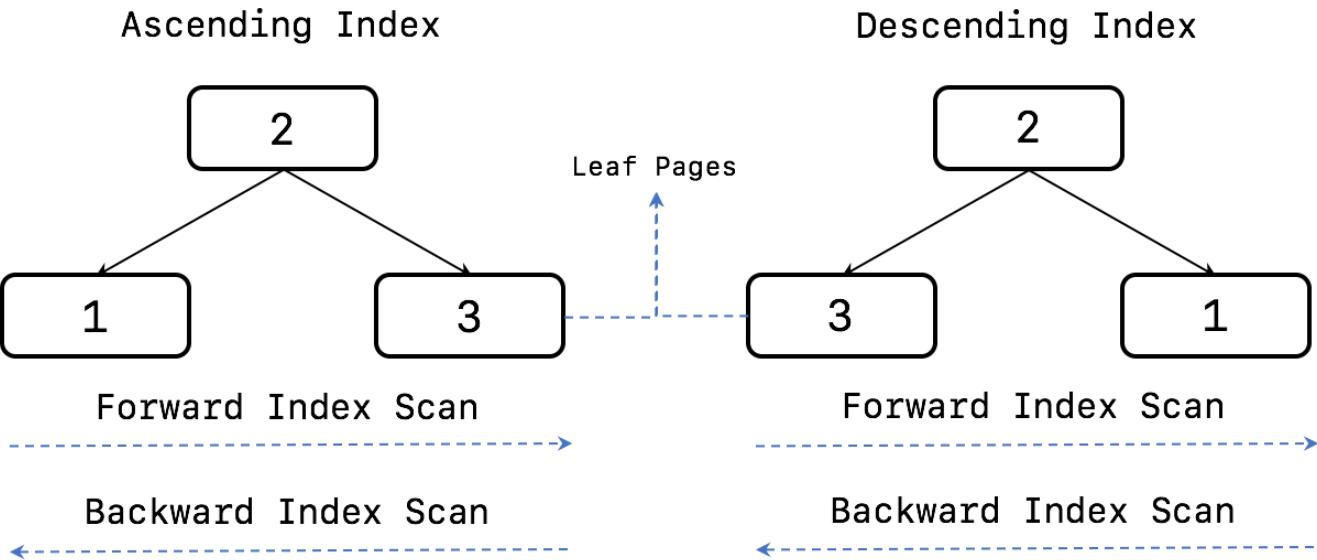
쿼리의 order by 처리나 min() 또는 max() 함수 등의 최적화가 필요한 경우에도 MySQL 옵티마이저는 인덱스의 읽기 방향을 전환해서 사용하도록 실행 계획을 만들어냅니다.

## 내림차순 인덱스

- first\_name 컬럼을 역순으로 정렬하는 요건만 있다면 다음 2개 인덱스 중에서 어떤 것을 선택하는 것이 좋을까? 아니면 두 인덱스 모두 동일한 성능을 보일까?

```
create index ix(firstName_asc) on employees (first_name asc);
create index ix(firstName_desc) on employees (first_name desc);
```

위 궁금증에 대한 답을 찾기 위해 MySQL 8.0부터 지원되는 내림차순 인덱스에 대해 조금 깊이 있게 살펴보자.



- 오름차순 인덱스(Asending index): 작은 값의 인덱스 키가 B-Tree의 왼쪽으로 정렬된 인덱스
- 내림차순 인덱스(Descending index): 큰 값의 인덱스 키가 B-Tree의 왼쪽으로 정렬된 인덱스
- 인덱스 정순 스캔 (Forward index scan): 인덱스 키의 크고 작음에 관계없이 인덱스 리프 노드의 왼쪽 페이지부터 오른쪽으로 스캔
- 인덱스 역순 스캔(Backward index scan): 인덱스 키의 크고 작음에 관계없이 인덱스 리프 노드의 오른쪽 페이지부터 왼쪽으로 스캔

- 이제 내림차순 인덱스의 필요성에 간단한 테스트 결과를 살펴보면서 알아보자.
- 간단한 테스트를 위해 다음과 같이 테스트용 테이블을 생성하고 대략 1천만 건 정도의 레코드를 준비해 보자.

The screenshot shows the MySQL Workbench interface. In the SQL editor, the following code is displayed:

```

48
49  create table t1
50  (
51      tid      int not null auto_increment,
52      table_name varchar(64),
53      column_name varchar(64),
54      ordinal_position int,
55      primary key (tid)
56  ) engine = InnoDB;
57
58
59  insert into t1
60  select tid, table_name, column_name, ordinal_position from information_schema.columns;
61
62  ✓ insert into t1
63  select tid, table_name, column_name, ordinal_position from t1;
64
65  ✓ select count(*) from t1;

```

In the Output tab, the result of the final query is shown:

count(*)
14749696

- 위 테이블을 풀 스캔하면서 정렬만 수행하는 쿼리를 실행해보면
- 두 쿼리는 테이블의 프라이머리 키를 정순 또는 역순으로 스캔하면서 마지막 레코드 1건만 반환한다.
- 첫 번째 쿼리는 tid 컬럼의 값이 가장 큰 레코드 1건을
- 그리고 두 번째 쿼리는 tid 컬럼의 값이 가장 작은 레코드 1건을 반환한다.
- 하지만 Limit...Offset 부분의 쿼리로 인해 실제 MySQL 서버는 테이블의 모든 레코드를 스캔해야 한다.

```

select * from t1 order by tid asc limit 14749696, 1; // 5초 걸림
select * from t1 order by tid desc limit 14749696, 1; // 7초 걸림

```

- 1,400여백만 건을 스캔하는 이정도 차이라고 생각할 수 있다. 하지만 비율로 따져 보면 역순 정렬 쿼리 가 정순 정렬 쿼리보다 30% 더 시간이 걸리는 것을 알 수 있다.
- 하나의 인덱스를 정순으로 읽느냐 또는 역순으로 읽느냐에 따라 이런 차이가 발생한다는 것은 쉽게 이해하기 어려울 수 있다.

- MySQL 서버의 InnoDB 스토리지 엔진에서 정순 스캔과 역순 스캔은 페이지(블록) 간의 양방향 연결 고리(Double Linked List)를 통해 전진(Forward) 이나 후진(Bakward)하느냐의 차이만 있지만
- 실제 내부적으로 InnoDB에서 인덱스 역순 스캔이 인덱스 정순 스캔에 비해 느릴 수 밖에 없는 두 가지 이유가 있다.

- 페이지 잠금이 인덱스 정순 스캔(Forward index scan)에 적합한 구조
- 페이지 내에서 인덱스 레코드가 단방향으로만 연결된 구조

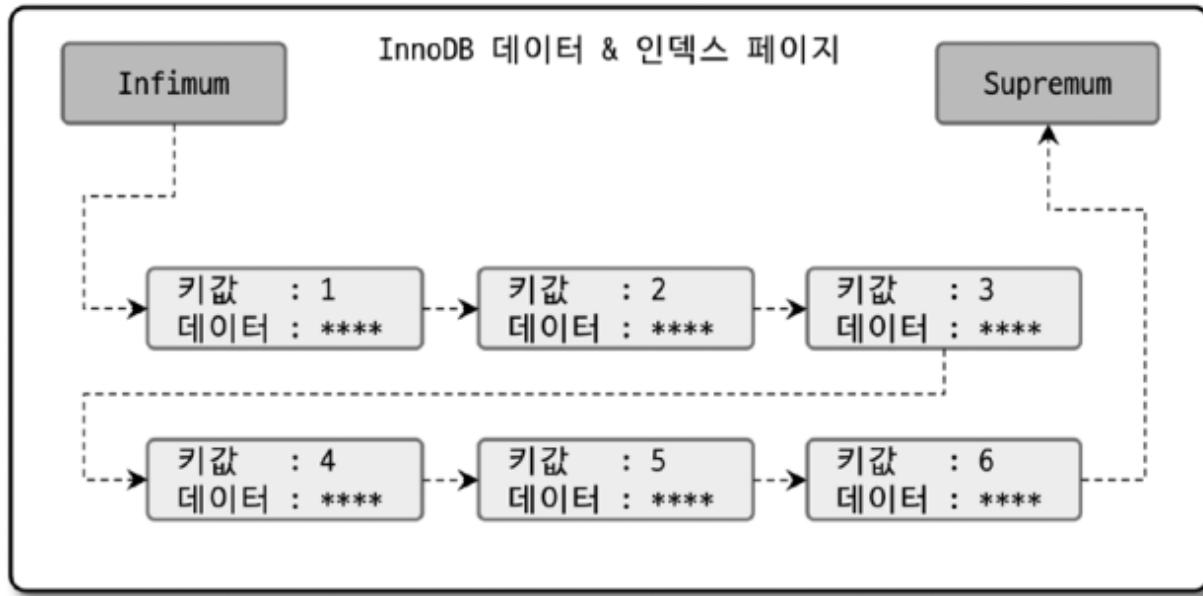


그림 8.16 InnoDB 페이지 내에서 레코드들의 연결

△ 참고

위 그림에서는 InnoDB 페이지 내부에서 레코드들이 정렬 순서대로 저장돼 있는 것처럼 표시돼 있지만 실제로 InnoDB 페이지는 힙처럼 사용되기 때문에 물리적으로 저장이 순서대로 배치되지는 않는다.

그리고 각 데이터 페이지나 인덱스 페이지의 엔트리는 키 값과 데이터를 가지는데, 인덱스의 루트 노드 또는 브랜치 노드라면 자식 노드의 주소를 가진다.

프라이머리 키에서 리프 노드의 "데이터"는 실제 레코드의 컬럼 값들이며, 세컨더리 인덱스 페이지에서는 프라이머리 키 값을 가진다.

- 내림차순과 오름차순 인덱스의 내부적인 차이로 인한 성능을 살펴봤습니다.
- 이제 서비스 요구에 맞게 정렬 순서의 인덱스를 선택해야 할지 살펴보면 일반적으로 인덱스를 Order By ...DESC 하는 쿼리가 소량의 레코드에 드물게 실행되는 경우라면 내림차순 인덱스를 굳이 고려할 필요는 없어 보입니다.

```
select * from tab where userid=? order by score desc limit 10
```

이 쿼리의 경우 아래 두 가지 인덱스 모두 적절한 선택이 될 수 있습니다.

- 오름차순 인덱스: INDEX(userid ASC, score ASC)
- 내림차순 인덱스: INDEX(userid DESC, score DESC)
- 하지만 위 쿼리가 많은 레코드를 조회하면서 빈번하게 실행된다면 오름차순 인덱스보다는 내림차순 인덱스가 더 효율적이라고 볼 수 있습니다.
- 또한 많은 쿼리가 인덱스의 앞쪽만 또는 뒤쪽만 집중적으로 읽어서 인덱스의 특정 페이지 임금이 병목이 될 것으로 예상된다면 쿼리에서 자주 사용되는 정렬 순서대로
- 인덱스를 생성하는 것이 임금 병목 현상을 완화하는데 도움이 될 것입니다.

## B-Tree 인덱스의 가용성과 효율성

- 다중 컬럼 인덱스에서 각 컬럼의 순서와 그 컬럼에 사용된 조건이 동등 비교('=')인지 아니면 크다(">") 또는 작다(" <") 같은 범위 조건인지에 따라 각 인덱스 컬럼의 활용 형태가 달라지며, 그 효율 또한 달라집니다.



```
select * from dept_emp where dept_no='d002' and emp_no >= 10114;
```

위 쿼리를 위해 dept\_emp 테이블에 각각 컬럼의 순서만 다른 두 가지 케이스로 인덱스를 생성했다고 가정하자.

- 케이스 A: INDEX(dept\_no, emp\_no)
- 케이스 B: INDEX(emp\_no, dept\_no)

케이스 A 인덱스는

```
dept_no='d002' AND emp_no >= 10144
```

인 레코드를 찾고 그 이후에는 dept\_no가 'd0002'가 아닐 때까지 인덱스를 그냥 쭉 읽기만 하면 된다.

- 이 경우에는 읽은 레코드가 모두 사용자가 원하는 결과임을 알 수 있다.
- 즉 조건을 만족하는 레코드가 5건이라고 할 때, 5건의 레코드를 찾는 데 꼭 필요한 5번의 비교 작업만 수행한 것이므로 상당히 효율적으로 인덱스를 이용한 것이다.
- 하지만 케이스 B인덱스는

```
emp_no >= 10144 AND dept_no='d002'
```

인 레코드를 찾고, 그 이후 모든 레코드에 대해 dept\_no가 'd002'인지 비교하는 과정을 거쳐야 합니다.

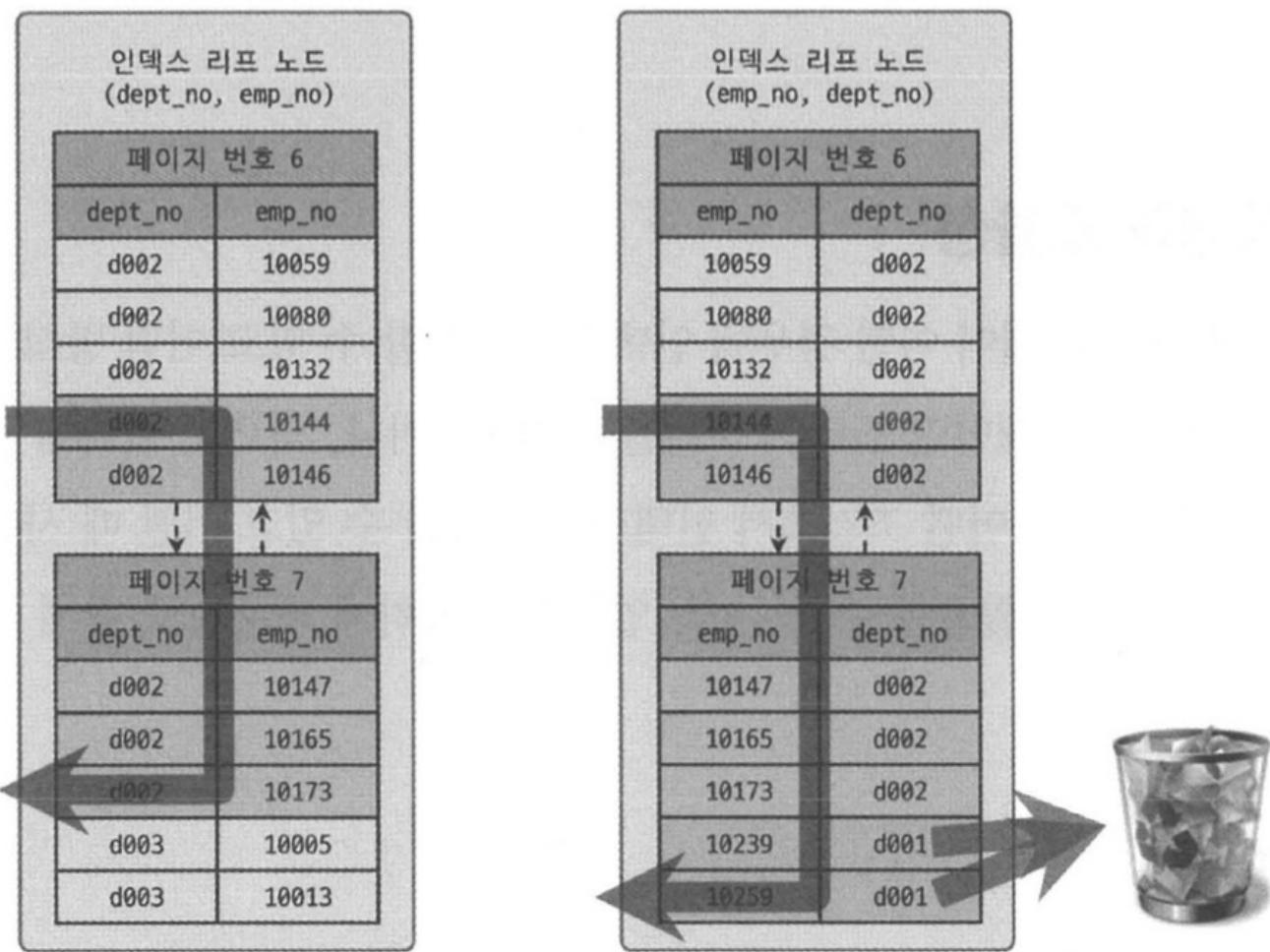


그림 8.17 인덱스의 칼럼 순서로 인한 쿼리 실행 내역의 차이

이처럼 인덱스를 통해 읽은 레코드가 나머지 조건에 맞는지 비교하면서 취사선택하는 작업을 '필터링'이라고 합니다.

케이스 B 인덱스에서는 최종적으로 dept\_no='d002' 조건을 만족(필터링)하는 레코드 5건을 가져옵니다. 이 경우에는 5건의 레코드를 찾기 위해 7번의 비교 과정을 거친 것입니다. (범위를 찾고 d002인지 확인) 이런 현상이 발생하는 이유는 이전 다중 컬럼 인덱스에서 설명한 다중 컬럼 인덱스의 정렬 방식 때문입니다. 다중 컬럼 인덱스의 정렬 방식은 인덱스의 N번째 키 값은 N-1번째 키 값에 대해서 다시 정렬되는 특성이 있습니다.

케이스 A 인덱스에서 2번 째 컬럼인 emp\_no는 비교 작업의 범위를 좁히는 데 도움을 줍니다.

하지만 케이스 B인덱스에서 2번째 컬럼인 dept\_no는 비교 작업의 범위를 좁히는 데 아무런 도움을 주지 못하고 단지 쿼리의 조건에 맞는지 검사하는 용도로만 사용됐다.

공식적인 명칭은 아니지만 케이스 A 인덱스에서의 두 조건과 같이 작업의 범위를 결정하는 조건을 '작업 범위 결정 조건'이라고 하고

케이스 B인덱스의 조건과 같이 비교 작업의 범위를 줄이지 못하고 단순히 거름종이 역할만 하는 조건을 필터링 조건 또는 체크 조건이라고 합니다.

결국 케이스 A인덱스에서 dept\_no 컬럼과 emp\_no 컬럼은 모두 작업 범위 결정 조건에 해당하지만, 케이스 B 인덱스에서는 emp\_no 컬럼만 작업 범위 결정 조건이고 dept\_no 컬럼은 필터링 조건으로 사용된 것입니다.

이처럼 작업 범위를 결정하는 조건은 많으면 많을수록 쿼리의 처리 성능을 높이지만 체크 조건은 많다고 해서 쿼리의 처리 성능을 높이지는 못하고 오히려 쿼리 실행을 더 느리게 만들 때가 많다.

## 인덱스의 가용성

B-Tree 인덱스의 특징은 왼쪽 값에 기준해서 오른쪽 값이 정렬돼 있다는 것입니다.

- 여기서 왼쪽이란 하나의 컬럼 내에서뿐만 아니라 다중 컬럼 인덱스의 컬럼에 대해서도 함께 적용됩니다.
- 케이스 A: INDEX(first\_name)
- 케이스 B: INDEX(dept\_no, emp\_no)

아래 그림에서 인덱스 키값의 정렬만 표현하지만 실제로 인덱스 키 값의 이런 정렬 특성은 빠른 검색의 전제 조건입니다.

즉 하나의 컬럼만으로 검색해도 값의 왼쪽 부분이 없으면 인덱스 레인지 스캔 방식의 검색이 불가능합니다. 또한 다중 컬럼 인덱스에서도 왼쪽 컬럼의 값을 모르면 인덱스 레인지 스캔을 사용할 수 없습니다. (실제로 b-tree 노드 같은 경우 왼쪽 값을 알 수 있도록 연결돼 있습니다.)

케이스 A의 인덱스가 지정된 employees 테이블에 대해 다음과 같은 쿼리가 어떻게 실행되는지 보겠습니다.



[그림 5-16] 왼쪽 값(Left-most)을 기준으로 정렬

```
select * from employees where first_name like '%mer';
```

위 쿼리는 인덱스 레인지 스캔 방식으로 인덱스를 이용할 수 없습니다.

그 이유는 `first_name` 컬럼에 저장된 값의 왼쪽부터 한 글자씩 비교해 가면서 일치하는 레코드를 찾아야 하는데, 조건절에 주어진 상수값 ('%mer')에는 왼쪽 부분이 고정되지 않았기 때문입니다.

따라서 정렬 우선순위가 낮은 뒷부분의 값만으로는 왼쪽 기준 정렬 기반의 인덱스인 B-tree에서는 인덱스의 효과를 얻을 수 없습니다.

케이스 B의 인덱스가 지정된 `dept_emp` 테이블에 대해 다음 쿼리가 어떻게 실행되는지 한번 살펴보자.

```
select * from dept_emp where emp_no >= 10144;
```

인덱스가 (`dept_no`, `emp_no`) 컬럼 순서대로 생성돼 있다면 인덱스의 선행 컬럼인 `dept_no` 조건 없이 `emp_no` 값으로만 검색하면 인덱스를 효율적으로 사용할 수 없습니다.

케이스 B의 인덱스는 다중 컬럼으로 구성된 인덱스이므로 `dept_no` 컬럼에 대해 먼저 정렬한 후, 다시 `emp_no` 컬럼값으로 정렬돼 있기 때문입니다.

## 가용성과 효율성 판단

기본적으로 B-Tree인덱스의 특성상 다음 조건에서는 사용할 수 없습니다. (여기서 사용할 수 없다는 것은 작업 범위 결정 조건으로 사용할 수 없다는 것을 의미하며, 경우에 따라서는 체크 조건으로 인덱스를 사용할 수는 있습니다)

```
# NOT-EQUAL로 비교된 경우 (<>, NOT IN, NOT BETWEEN, IS NOT NULL)
WHERE column <> 'N'
where column not in (10, 11,12)
where column is not null

# Like '%??%'(앞부분이 아닌 뒷부분 일치) 형태로 문자열 패턴이 비교된 경우
where column like '%'
where column like '_'
where column like '%%'

# 스토어드 함수나 다른 연산자로 인덱스 컬럼이 변형된 후 비교된 경우
where substring(column, 1,1) ='x'
where dayofmonth(column) = 1

# 데이터 타입이 서로 다른 비교(인덱스 컬럼의 타입을 변환해야 비교가 가능한 경우)
where char_column = 10

#문자열 데이터 타입의 콜레이션이 다른 경우
```

```
where utf8_bin_char_column = euckr_bin_char_column
```

다른 일반적인 DBMS에서는 Null값이 인덱스에 저장되지 않지만 MySQL에서는 Null값도 인덱스에 저장됩니다.

다음과 같은 where 조건도 작업 범위 결정 조건으로 인덱스를 사용합니다.

```
Database changed
mysql> explain select * from employees where first_name like '%mer';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | employees | NULL | ALL | NULL | NULL | NULL | NULL | 300141 | 11.11 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> explain select * from employees where first_name like 'mer%';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | employees | NULL | range | ix_firstname | ix_firstname | 58 | NULL | 1 | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> explain select * from employees where first_name is null;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | NULL | Impossible WHERE |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

다중 컬럼으로 만들어진 인덱스는 조건에 따라 사용여부가 나누어집니다.

```
index ix_test (column_1, column2, column_3,..., column_n)
```

위 인덱스를 예시로 들면

- 작업 범위 결정 조건으로 인덱스를 사용하지 못하는 경우
  - column\_1 컬럼에 대한 조건이 없는 경우
  - column\_1 컬럼의 비교 조건이 위의 인덱스 사용 불가 조건 중 하나인 경우
- 작업 범위 결정 조건으로 인덱스를 사용하는 경우(i는 2보다 크고 n보다 작은 임의의 값을 의미)
  - column1 ~column(i-1) 컬럼까지 동등 비교 형태로 비교
  - column\_i 컬럼에 대해 다음 연산자 중 하나로 비교
    - 동등 비교 ("=" 또는 in)
    - 크다 작다 형태
    - Like로 좌측 일치 패턴 (LIKE '승환%')

위 두 가지 조건을 모두 만족하는 쿼리는 column1 부터 column\_i 까지는 작업 범위 결정 조건으로 사용되고, column(i+1)부터 column\_n까지의 조건은 체크 조건으로 사용됩니다.

그 외 추가적인 예시는 252p 참고

## R-Tree 인덱스

MySQL의 공간 인덱스라는 말을 한 번쯤 들어본 적이 있을 것입니다 (들어본적 없습니다..)

공간 인덱스는 R-Tree 인덱스 알고리즘을 이용해 2차원의 데이터를 인덱싱하고 검색하는 목적의 인덱스입니다.

기본적인 내부 메커니즘은 B-Tree와 흡사합니다.

B-Tree는 인덱스를 구성하는 컬럼의 값이 1차원의 스칼라 값인 반면 R-Tree는 2차원의 공간 개념 값입니다. 최근 GPS나 지도 서비스를 내장하는 스마트 폰이 대중화되면서 SNS 서비스가 GIS와 GPS에 반을 둔 서비스로 확장되고 있습니다.

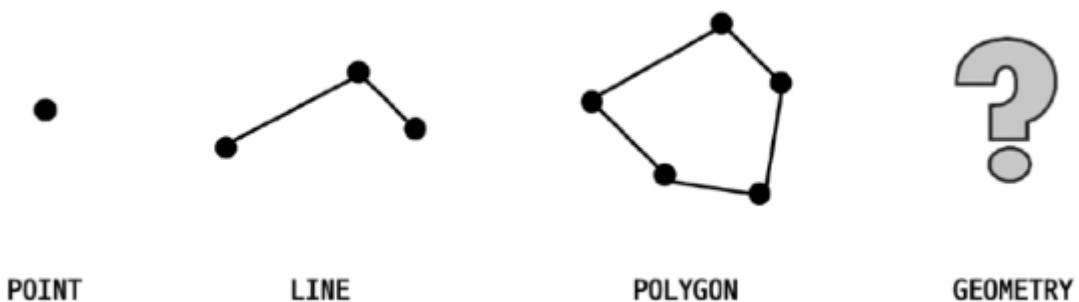
MySQL의 공간 확장을 이용하면 위치 기반 서비스를 간단하게 구현할 수 있습니다.

MySQL 공간 확장에는 다음과 같이 크게 세 가지의 기능이 포함돼 있습니다.

- 공간 데이터를 저장할 수 있는 데이터 타입
- 공간 데이터의 검색을 위한 공간 인덱스
- 공간 데이터의 연산 함수

## 구조 및 특성

- MySQL은 공간 정보의 저장 및 검색을 위해 여러가지 기하학적 도형 정보를 관리할 수 있는 데이터 타입을 제공합니다.



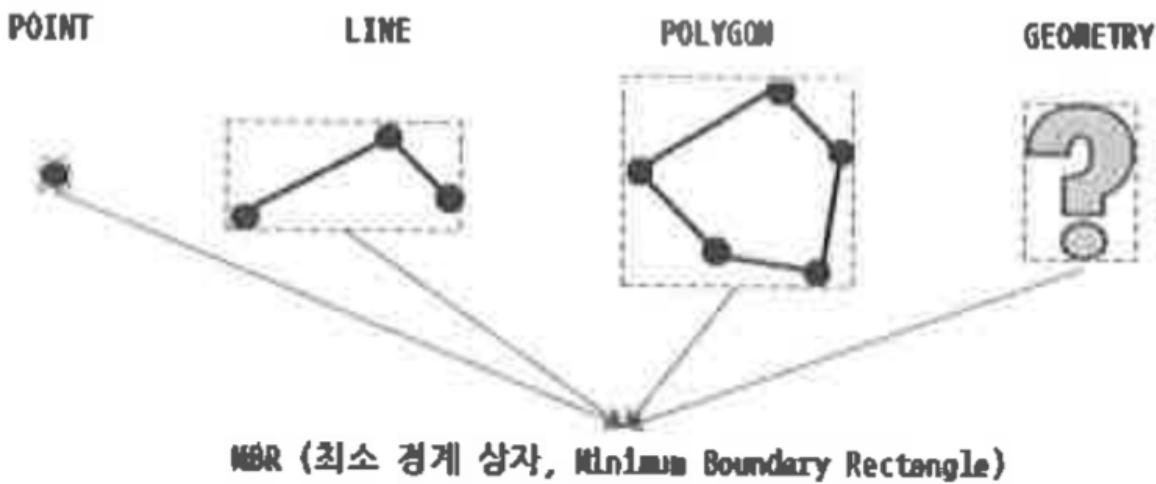
마지막에 있는 GEOMETRY 타입은 나머지 3개의 타입의 슈퍼 타입으로, Point와 Line, Polygon 객체를 모두 저장할 수 있습니다.

공간 정보의 검색을 위한 R-Tree 알고리즘을 이해하기 위해서 MBR이라는 개념을 알아야 합니다.

아래 그림은 위 그림에서 예시로 든 도형들의 MBR을 나타냅니다.

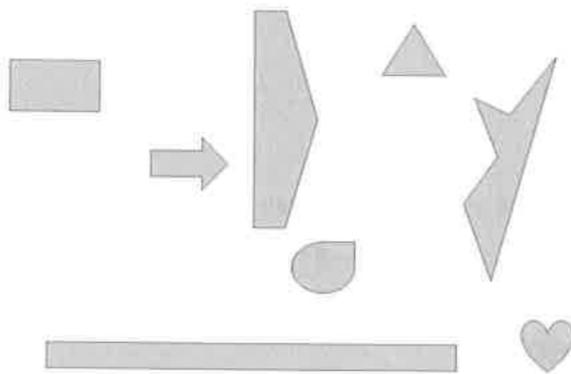
여기서 MBR은 **Minimum Bounding Rectangle**의 약자로 해당 ⌂형을 감싸는 최소 크기의 사각형을 의미합니다.

이런 사각형들의 포함 관계를 B-Tree 형태로 구현한 것이 R-Tree 인덱스입니다.

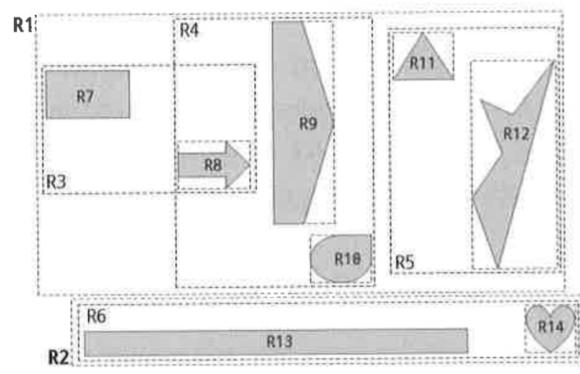


[그림 5-19] 최소 경계 상자(MBR, Minimum Bounding Rectangle)

아래 R-Tree의 구조를 보면 다음과 같은 도형(공간 데이터)가 있다고 가정해보겠습니다.



[그림 5-20] 공간(Spatial) 데이터



[그림 5-21] 공간(Spatial) 데이터의 MBR

여기에는 포시되지 않았지만 단순히 X,Y 좌표만 있는 포인트 데이터 또한 하나의 도형 객체가 될 수 있습니다.  
위 그림에서 도형들을 MBR을 3개월 레벨로 나눠서 그려보면 다음과 같습니다.

- 최상위 레벨 R1,R2
- 차상위 레벨 R3,R4,R5,R6
- 최하위 레벨 R7~R14

최하위 레벨의 MBR은 각 도형 데이터의 MBR을 의미합니다.

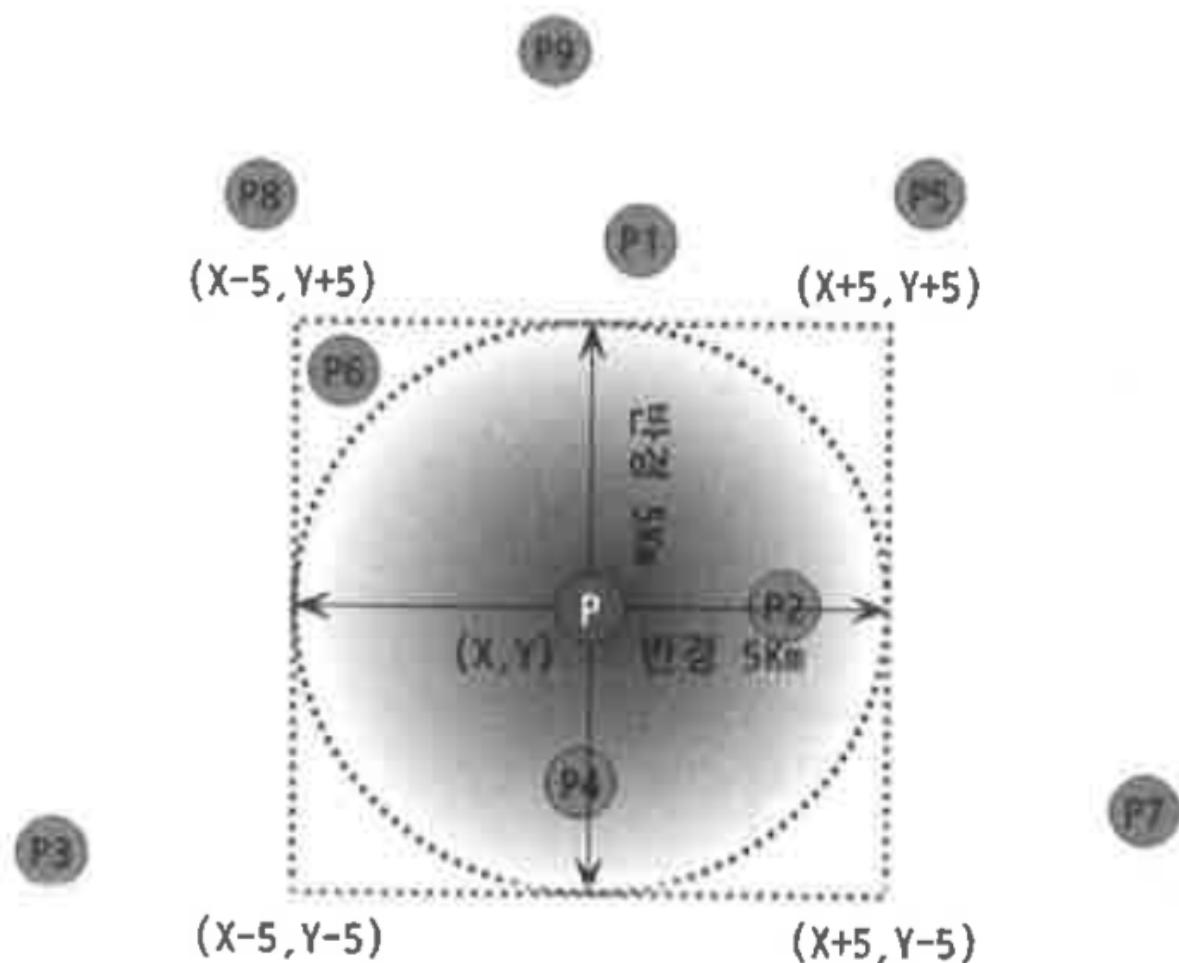
그리고 차상위 레벨의 MBR은 중간 크기의 MBR입니다.

위 예제에서 최상위 MBR은 R-TREE의 루트 노드에 저장되는 정보이며 차상위 그룹 MBR은 R-Tree의 브랜치 노드가 됩니다.

최하위 레벨은 리프 노드에 저장되는 특징을 가집니다

## R-Tree 인덱스의 용도

- R-Tree는 앞에서 언급한 MBR 정보를 이용해 B-Tree형태로 인덱스를 구축하므로 Rectangle의 'R'과 B-Tree의 Tree를 섞어서 R-Tree라는 이름이 붙어졌으며 공간 인덱스라고 합니다.
- 일반적으로 WGS84기준의 위도, 경도 좌표 저장에 주로 사용됩니다.
- 하지만 위도 경도 좌표뿐 아니라 CAD/CAM 소프트웨어 또는 회로 디자인 등과 같이 좌표 시스템에 기반을 둔 정보에 대해서 모두 적용 가능합니다.
- 아래 그림에서 알 수 있듯이 R-Tree는 각 도형의 포함 관계를 이용해 만들어진 인덱스입니다.
- 따라서 ST\_Contains 또는 ST\_Within 등과 같은 포함 관계를 비교하는 함수로 곱색을 수행하는 경우에만 인덱스를 이용할 수 있습니다.
- 대표적으로 현재 사용자의 위치로부터 반경 5KM이내의 음식점 검색 등과 같은 검색에서 사용됩니다.



[그림 5-23] 특정 지점을 기준으로 사각 박스 이내의 위치를 검색

위 그림에서 가운데 위치한 P가 기준점입니다.

기준점으로부터 반경 5KM이내의 점들을 검색하려면 우선 사각 점선의 상자에 포함되는 점들을 검색하면 됩니다.

여기서 ST\_Contains나 St\_Within연산은 사각형 박스와 같은 다각형으로만 연산할 수 있으므로 반경 5KM를 그리는 원을 포함하는 최소 사각형으로 포함 관계 비교를 수행한 것이다.

P6은 기준점 P로부터 반경 5km 이상 떨어져 있지만 최소 사각형 내에는 포함된다.

P6을 빼고 결과를 조회하려면 조금 더 복잡한 비교가 필요하는데 P6를 결과에 포함해도 무방하면 다음 쿼리와 같이 St\_Contains나 St\_Within비교만 수행하는 것이 좋습니다.

```
select * from tb_location where st_contains(사각 상자, px);
select * from tb_location where st_within(px,사각 상자);
```

st\_contains함수와 st\_within함수는 거의 동일한 비교를 수행하지만 두 함수의 파라미터는 반대로 사용해야 합니다.

## 전문 검색 인덱스

지금까지 살펴본 인덱스 알고리즘은 일반적으로 크지 않은 데이터 또는 이미 키워드화한 작은 값에 대한 인덱싱 알고리즘이었습니다.

대표적으로 MySQL의 B-Tree 인덱스는 실제 컬럼의 값이 1MB이더라도 1MB 전체의 값을 인덱스 키로 사용하는 것이 아닌 3072바이트 (InnoDB 기준) 까지만 잘라서 인덱스 키로 사용합니다.

또한 B-Tree 인덱스의 특성에서도 알아봤듯이 전체 일치 또는 좌측 일부 일치와 같은 검색만 가능합니다.

문서의 내용 전체를 인덱스화해서 특정 키워드가 포함된 문서를 검색하는 전문 (Full Text) 검색에는 스토리지 엔진에서 제공하는 일반적인 B-Tree인덱스를 사용할 수 없습니다.

문서 전체에 대한 분석과 검색을 위한 인덱싱 알고리즘을 전문 검색(Full Text Search)인덱스라고 하는데, 전문 검색 인덱스는 일반화된 기능의 명칭이지 전문 검색 알고리즘의 이름을 지칭하는 것은 아닙니다.

## 인덱스 알고리즘

전문 검색에서는 문서 본문의 내용에서 사용자가 검색하게 될 키워드를 분석해 내고, 빠른 검색용으로 사용할 수 있게 이러한 키워드로 인덱스를 구축합니다.

전문 검색 인덱스는 문서의 키워드를 인덱싱하는 기법에 따라 크게

- 단어의 어근 분석과
- n-gram 분석

알고리즘으로 구분할 수 있습니다. 예전에는 구분자도 하나의 알고리즘처럼 생각됐지만 8점대 부터는 구 분자 방식은 이미 어근 분석과 n-gram 알고리즘에 함께 포함됐습니다.

## 어근 분석 알고리즘

MySQL 서버의 전문 검색 인덱스는 두 가지 중요한 과정을 거쳐서 색인 작업이 수행됩니다.

- 불용어 (Stop Word) 처리
- 어근 분석(Stemming)

불용어 처리는 검색에서 별 가치가 없는 단어를 모두 필터링해서 제거하는 작업을 의미합니다. 불용어의 개수는 많지 않기 때문에 구현한 코드에서 모두 상수로 정의해서 사용하는 경우가 많고, 유연성을 위해 불용어 자체를 데이터베이스화해서 사용자가 추가하거나 삭제할 수 있게 구현하는 경우도 있습니다. 현재 MySQL 서버는 불용어가 소스코드에 정의돼 있지만, 이를 무시하고 사용자가 별도로 불용어를 정의할 수 있는 기능을 제공합니다.

어근 분석은 검색어로 선정된 단어의 뿌리인 원형을 찾는 작업입니다.

MySQL 서버에서는 오픈소스 형태소 분석 라이브러리인 MeCab을 플러그인 형태로 사용할 수 있게 지원하는데

한글이나 일본어의 경우 영어와 같이 단어의 변형 자체는 거의 없기 때문에 어근 분석보다는 문장의 형태소를 분석해서 명사와 조사를 구분하는 기능이 더 중요한 편입니다.

MeCab을 제대로 사용하려면 단어 사전이 필요하며, 문장을 해체해서 각 단어의 품사를 식별할 수 있는 문장의 구조 인식이 필요합니다. 문장의 구조 인식을 위해서는 실제 언어의 샘플을 이용해 언어를 학습하는 과정이 필요한데, 이 과정은 상당한 시간이 필요한 작업입니다.

## n-gram 알고리즘

n-gram 알고리즘을 사용하는 이유는 전문적인 검색 엔진을 고려하지 않고 범용적으로 적용하기 위해서입니다.

형태소 분석이 문장을 이해하는 알고리즘이라면 n-gram은 단순히 키워드를 검색해내기 위한 인덱싱 알고리즘이라고 할 수 있습니다.

n-gram이란 본문을 무조건 몇 글자씩 잘라서 인덱싱하는 방법입니다.

형태소 분석보다는 알고리즘이 단순하고 국가별 언어에 대한 이해와 준비 작업이 필요 없는 반면, 만들어진 인덱스의 크기는 상당히 큰 편입니다.

n-gram에서 n은 인덱싱할 키워드의 최소 글자 수를 의미하는데, 일반적으로 2글자 단위로 키워드를 쪼개서 인덱싱하는 2-gram(Bi-gram)방식이 많이 사용됩니다.

- MySQL 서버의 n-gram 알고리즘을 이해하기 위해서 2-gram 알고리즘으로 다음 문장의 토큰을 분리해 보겠습니다.

To be or not to be. That is the question

각 단어는 다음과 같이 띄어쓰기와 마침표를 기준으로 10개의 단어로 구분되고 2글자씩 중첩해서 토큰으로 분리됩니다.

주의해야 할 것은 각 글자가 중첩해서 2글자씩 토큰으로 구분됐다는 것입니다.

그래서 10글자 단어라면 그 단어는 2-gram 알고리즘에서는 (10-1)개의 토큰으로 구분됩니다.

이렇게 각 토큰을 인덱스에 저장하기만 하면 됩니다.

이때 중복된 토큰은 하나의 인덱스 엔트리로 병합되어 저장됩니다.

단어	bi-gram(2-gram) 토큰						
To	To						
be	be						
or	or						
not	no	ot					
to	to						
be	be						
That	Th	ha	at				
is	is						
the	th	he					
question	qu	ue	es	st	ti	io	on

MySQL 서버는 이렇게 생성된 토큰들에 대해 불용어를 걸러내는 작업을 수행하는데, 이때 불용어와 동일하거나 불용어를 포함하는 경우 거릅니다.

기본적으로 MySQL 서버에 내장된 불용어는 `information_schema`에서 확인할 수 있습니다.

The screenshot shows a MySQL Workbench interface. In the top right, a query window displays the command: `select * from information_schema.INNODB_FT_DEFAULT_STOPWORD;`. The result set, titled "information\_schema.innodb\_ft\_default\_stopword", is shown in a table with one column labeled "value". The table contains 36 rows, each numbered from 1 to 15, listing common English stop words.

	value
1	a
2	about
3	an
4	are
5	as
6	at
7	be
8	by
9	com
10	de
11	en
12	for
13	from
14	how
15	i

information\_schema.ft\_default\_stopword 테이블에 등록된 불용어의 목록은 다음과 같습니다.

```
a,about,an,are,as,at,be,by,com,de,en,for,from,how,i,in,is,it,la,of,on,or,tha  
t,the,this,to,was,what,when,where,who,will,with,und,the,www
```

MySQL 서버에서 실제 저장되는 엔트리는 다음과 같으며 결과적으로 MySQL 서버는 다음 표의 출력 컬럼에 표시된 것들만 전문 검색 인덱스에 등록합니다.

(262p 참고)

물론 전문 검색을 더 빠르게 하기 위해 2단계 인덱싱(프론트 혹은 백엔드 인덱스)과 같은 방법도 있지만 MySQL 서버는 B-Tree에 저장하며 성능 향상을 위한 Merge-Tree 같은 기능을 가지고 있긴 합니다.

## 불용어 변경 및 삭제

- 앞서 살펴본 n-gram의 토큰 파싱 및 불용어 처리 예시에서 결과를 보며 'ti, at, ha' 같은 토큰들은 'a, i' 철자가 불용어로 등록돼 있기 때문에 모두 걸러져서 버려집니다.
- 실제로 이 같은 불용어 처리는 사용자에게 도움이 되기보다는 혼란스럽게 하는 기능입니다.
- 그래서 불용어 처리를 사용하지 않거나 직접 불용어를 등록하는 방법을 권장합니다.

## 전문 검색 인덱스의 불용어 처리 무시

- 불용어 처리를 무시하는 방법은 두 가지가 있습니다.
- 스토리지 엔진에 관계없이 MySQL 서버의 모든 전문 검색 인덱스에 대해 불용어를 완전히 제거하는 방법
  - InnoDB 스토리지 엔진을 사용하는 테이블의 전문 검색 인덱스에 대해서만 불용어 처리를 무시하는 방법

첫 번째 방법은 my.cnf 의 ft\_stopword\_file 시스템 변수에 빈 문자열을 설정하면 됩니다.

```
ft_stopword_file=''
```

- ft\_stopword\_file 시스템 변수는 MySQL 서버의 내장 불용어를 비활성화할 때도 사용할 수 있지만 사용자 정의 불용어를 적용할 때도 사용할 수 있습니다.
- 사용자가 직접 정의한 불용어 목록을 저장한 파일의 경로를 ft\_stopword\_file 시스템 변수에 설정하면 해당 경로의 파일에서 불용어 목록을 가져와 적용합니다.

두 번째 방법은 InnoDB 테이블의 전문 검색 인덱스의 불용어 처리를 무시하기 위해 innodb\_ft\_enable\_stopword 시스템 변수를 OFF로 설정하는 것입니다.

이 경우 MySQL 서버의 다른 스토리지 엔진을 사용하는 테이블은 여전히 내장 불용어 처리를 사용합니다. innodb\_ft\_enable\_stopword는 동적인 시스템 변수이므로 MySQL 서버가 실행 중인 상태에서도 변경할 수 있습니다.

```
set global innodb_ft_enable_stopword=off;
```

## 사용자 정의 불용어 사용

앞서 말한대로 사용자 정의 불용어를 사용하는 방법은 두 가지가 있습니다.

- 불용어 목록을 파일로 저장하고, MySQL 서버 설정 파일에서 파일의 경로를 다음과 같이 ft\_stopword\_file로 설정
- InnoDB 스토리지 엔진에서 사용하는 방법으로 불용어 테이블을 생성하고, innodb\_ft\_server\_stopword\_table 시스템 변수에 불용어 테이블을 설정하는 방법입니다.

두 번째 방법은 불용어 목록을 변경한 이후 전문 검색 인덱스가 생성돼야만 변경된 불용어가 적용된다는 점을 주의해야 합니다.

```
create table my_stopword(value varchar(30)) engine = INNODB;
insert into my_stopword(value) values('MySQL');

set global innodb_ft_enable_stopword='mydb/my_stopword';

Alter table tb_bi_gram add fulltext fx_title_body(title, body) with parser ngram;
```

## 전문 검색 인덱스의 가능성

전문 검색 인덱스를 사용하려면 다음 두 가지 조건을 필수적으로 갖춰야 합니다

1. 쿼리 문장이 전문 검색을 위한 문법 (Match ... against ...)을 사용
2. 테이블이 전문 검색 대상 컬럼에 대해서 전문 인덱스 보유

다음과 같이 테이블의 컬럼에 전문 검색 인덱스를 생성했다고 가정했을 때

```
create table tb_test(
    doc_id int,
    doc_body text,
    primary key (doc_id),
    fulltext key fx_docbody (doc_body)with parser ngram
) engine = innodb;

explain select * from tb_test where doc_body like '%애플%';

explain select * from tb_test where match(doc_body) AGAINST('애플' in boolean mode)
```

검색 쿼리로도 동일한 결과를 얻을 수 있지만 match against 구분을 사용해야 전문 검색 인덱스가 적용된다.

## 함수 기반 인덱스

일반적으로 인덱스는 컬럼의 값 일부(컬럼의 값 앞부분) 또는 전체에 대해서만 인덱스 생성이 허용됩니다. 하지만 때로는 컬럼의 값을 변형해서 만들어진 값에 대해 인덱스를 구축해야 할 때도 있는데 이러한 경우 함수 기반의 인덱스를 활용하면 됩니다.

### ▲ 참고

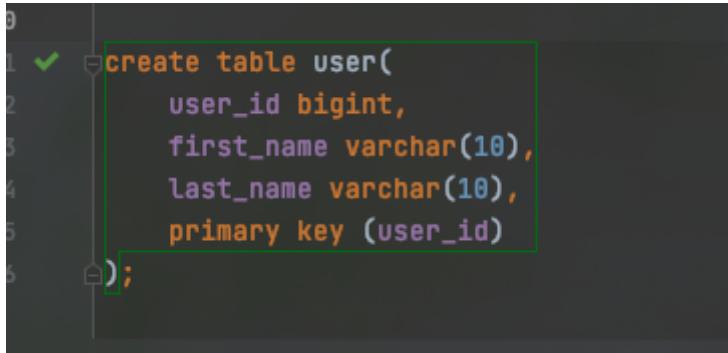
MySQL 서버는 8.0 버전부터 함수 기반 인덱스를 지원하기 시작함

MySQL 서버에서 함수 기반 인덱스를 구현하는 방법은 두 가지가 있습니다.

1. 가상 컬럼을 이용한 인덱스
2. 함수를 이용한 인덱스

MySQL 서버의 함수 기반 인덱스는 인덱싱할 값을 계산하는 과정의 차이만 있을 뿐, 실제 인덱스의 내부적인 구조 및 유지관리 방법은 B-Tree인덱스와 동일합니다.

## 가상 컬럼을 이용한 인덱스



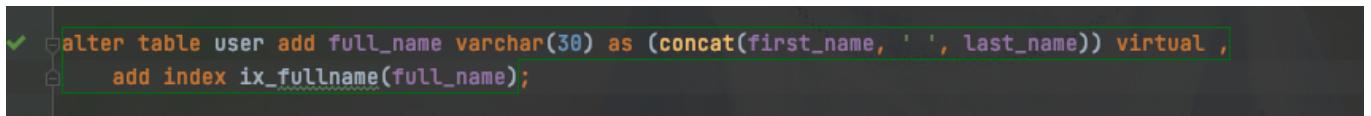
```
1 ✓ create table user(
2     user_id bigint,
3     first_name varchar(10),
4     last_name varchar(10),
5     primary key (user_id)
6 );
```

사용자 정보를 저장하는 테이블이 있고 first\_name과 last\_name을 합쳐서 검색하는 요구사항을 가정하겠습니다.

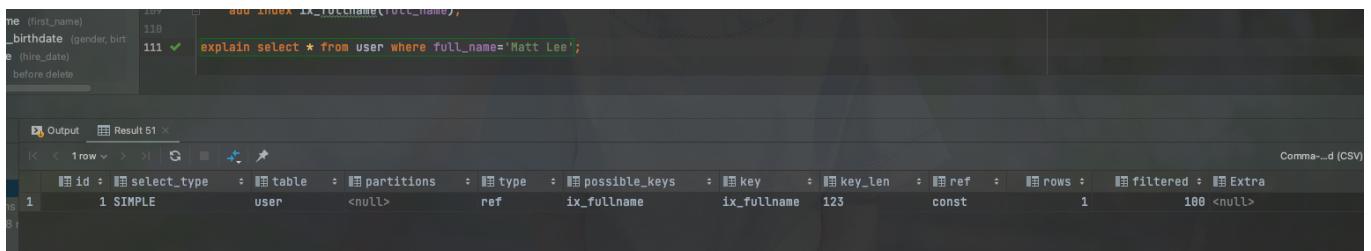
MySQL 서버에서는 full\_name이라는 컬럼을 추가하고 모든 레코드에 대해 full\_name을 업데이트하는 작업을 거쳐야 합니다.

그렇게 해야 full\_name 컬럼에 대해서 인덱스를 생성할 수 있습니다.

8.0부터는 아래와 같이 가상 컬럼에 인덱스를 생성할 수 있습니다.



```
✓ alter table user add full_name varchar(30) as (concat(first_name, ' ', last_name)) virtual ,
    add index ix_fullname(full_name);
```



```
1 ✓ explain select * from user where full_name='Matt Lee';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	<null>	ref	ix_fullname	ix_fullname	123	const	1	100	<null>

possible\_keys를 보면 ix\_fullname을 사용하고 있는 것을 확인할 수 있습니다.

가상 컬럼이 virtual이나 stored 옵션 중 어떤 옵션으로 생성됐든 관계없이 해당 가상 컬럼에 인덱스를 생성할 수 있습니다.

단 가상 컬럼에 인덱스를 생성하는 것은 테이블에 새로운 컬럼을 추가하는 것과 같은 효과를 내기 때문에 실제 테이블의 구조가 변경된다는 단점이 있습니다.

## 함수를 이용한 인덱스

8.0버전부터는 테이블 구조를 변경하지 않고 함수를 직접 사용하는 인덱스를 생성할 수 있게 됐습니다.

```
create table user(
    user_id bigint,
    first_name varchar(10),
    last_name varchar(10),
    primary key (user_id),
    index ix_fullname ((concat(first_name, ' ', last_name)))
);
```

함수를 직접 사용하는 인덱스는 테이블의 구조는 변경하지 않고, 계산된 결괏값의 검색을 빠르게 만들어줍니다.

함수 기반 인덱스를 제대로 활용하려면 반드시 조건절에 함수 기반 인덱스에 명시된 표현식이 그대로 사용돼야 합니다.

함수 생성 시 명시된 표현식과 쿼리의 where 조건절에 사용된 표현식이 다르면 MySQL 옵티마이저는 다른 표현식으로 갖누해서 함수 기반 인덱스를 사용할 수 없습니다. (중요)

```
explain select * from user where concat(first_name, ' ', last_name)='Matt Lee'  
explain select * from user where concat(first_name, '-', last_name)='Matt Lee'
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	<null>	ref	ix_fullname	ix_fullname	87	const	1	100	<null>

```
explain select * from user where concat(first_name, ' ', last_name)='Matt Lee'  
explain select * from user where concat(first_name, '-', last_name)='Matt Lee'
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	<null>	ALL	<null>	<null>	<null>	<null>	1	100	Using where

## 멀티 밸류 인덱스

전문 검색 인덱스를 제외한 모든 인덱스는 인덱스 키와 데이터 레코드는 1:1의 관계를 가집니다.

멀티 밸류 인덱스는 하나의 데이터 레코드가 여러 갑의 키 값을 가질 수 있는 형태의 인덱스로 일반적인 RDBMS를 기준으로 정규화에 위배되는 형태입니다.

이런 멀티 밸류 인덱스가 등장한 이유는 최근 RDBMS들이 JSON 데이터 타입을 지원하기 시작하면서 JSON의 배열 타입 필드에 저장된 원소들에 대한 인덱스 요건이 발생한 것에 비롯됐습니다.

JSON 포맷으로 데이터를 저장하는 MongoDB는 처음부터 이런 형태의 인덱스를 지원하고 있었지만 MySQL 서버는 멀티 밸류 인덱스에 대한 지원없이 JSON 타입의 컬럼만을 지원했습니다.

하지만 배열 형태에 대한 인덱스 생성이 되지 않아서 MongoDB의 기능과 많이 비교되곤 했습니다.

현재는 8.0 버전으로 업그레이드 되면서 MySQL 서버의 JSON관리 기능은 태생적으로 JSON을 사용했던 MongoDB에 비해서도 부족함이 없는 상태가 됐습니다.

예시로 다음과 같이 신용정보 점수를 배열로 JSON 타입 컬럼에 저장하는 테이블을 가정해보겠습니다.

```
128
129  create table user
130  (
131      user_id      bigint,
132      first_name   varchar(10),
133      last_name    varchar(10),
134      primary key (user_id),
135      credit_info  JSON,
136
137      index mx_creditscores (CAST(credit_info -> '$.credit_scores' as unsigned Array))
138  )
139  insert into user values (1, 'Matt', 'Lee', '{"credit_scores": [360, 353, 351]})_
140
141  select * from user;
```

The screenshot shows the MySQL Workbench interface. At the top, there is a code editor window with the above SQL script. Below it is a results grid titled "employees.user" showing one row of data:

user_id	first_name	last_name	credit_info
1	Matt	Lee	{"credit_scores": [360, 353, 351]}

멀티 벨류 인덱스를 활용하기 위해서는 일반적인 조건 방식이 아닌 다음 함수를 이용해서 검색해야 옵티마이저가 인덱스를 활용한 실행 계획을 수립합니다.

- member of()
- json contains()
- json overlaps()

```
mysql> select * from user where 360 member of(credit_info -> '$.credit_scores');
+-----+-----+-----+
| user_id | first_name | last_name | credit_info
+-----+-----+-----+
|     1   | Matt       | Lee       | {"credit_scores": [360, 353, 351]}
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from user where 360 member of(credit_info -> '$.credit_scores')
-> ;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | partitions | type | possible_keys | key        | key_len | ref   | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|  1  | SIMPLE      | user   | NULL       | ref  | mx_creditscores | mx_creditscores | 9       | const |    1 |   100.00 | Using where
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

멀티 벨류 인덱스를 활용해 실행 계획이 만들어진 것을 확인할 수 있습니다.

## 클러스터링 인덱스

- [클러스터링이란?](#)
  - 클러스터링이란 ‘무리를 이룬다’는 뜻으로, 서로 유사한 속성을 갖는 데이터를 같은 군집으로 묶어 주는 작업을 의미함

MySQL서버에서 클러스터링은 테이블의 레코드를 비슷한 것들끼리 묶어서 저장하는 형태로 구현되는데 이는 주로 비슷한 값을 동시에 조회하는 경우가 많다는 점에서 착안한 것입니다.

▲ 주의

InnoDB 스토리지엔진만 클러스터링 인덱스를 지원함

## 클러스터링 인덱스

클러스터링 인덱스는 테이블의 프라이머리 키에 대해서만 적용됩니다. 그렇기에 프라이머리 키 값이 비슷한 레코드끼리 묶어서 저장하는 것이라고 설명할 수 있습니다.

단 프라이머리 키 값에 의해 레코드의 저장 위치가 바뀌는데 이렇게 클러스터링된 테이블은 프라이머리 키 값 자체에 대한 의존도가 상당히 크기 때문에 신중히 결정해야 합니다.

클러스터링 인덱스는 프라이머리 키 값에 의해 레코드의 저장 위치가 결정되므로 사실 인덱스 알고리즘이라기보다 테이블 레코드의 저장 방식이라고 볼 수 있습니다.

그래서 "클러스터링 인덱스"와 "클로스터링 테이블"은 동의어로 사용되기도 합니다.

또한 클러스터링 기준이 되는 프라이머리 키는 클러스터링 키라고도 표현합니다.

일반적으로 InnoDB와 같이 항상 클러스터링 인덱스로 저장되는 테이블은 프라이머리 키 기반의 검색이 매우 빠르며, 레코드의 저장이나 프라이머리 키의 변경이 상대적으로 느립니다.

▲주의

일반적으로 B-Tree인덱스도 키 값으로 이미 정렬되어 저장됩니다.

이 또한 어떻게 보면 인덱스의 키 값으로 클러스터링 된 것으로 생각할 수 있습니다.

하지만 이러한 일반적인 B-Tree인덱스를 클러스터링 인덱스라고 부르지 않습니다.

테이블의 레코드가 프라이머리 키 값으로 정렬되어 저장된 경우만 클러스터링 인덱스 또는 클러스터링 테이블이라고 합니다.

아래는 클러스터링 테이블의 특성을 나타낸 구조입니다.

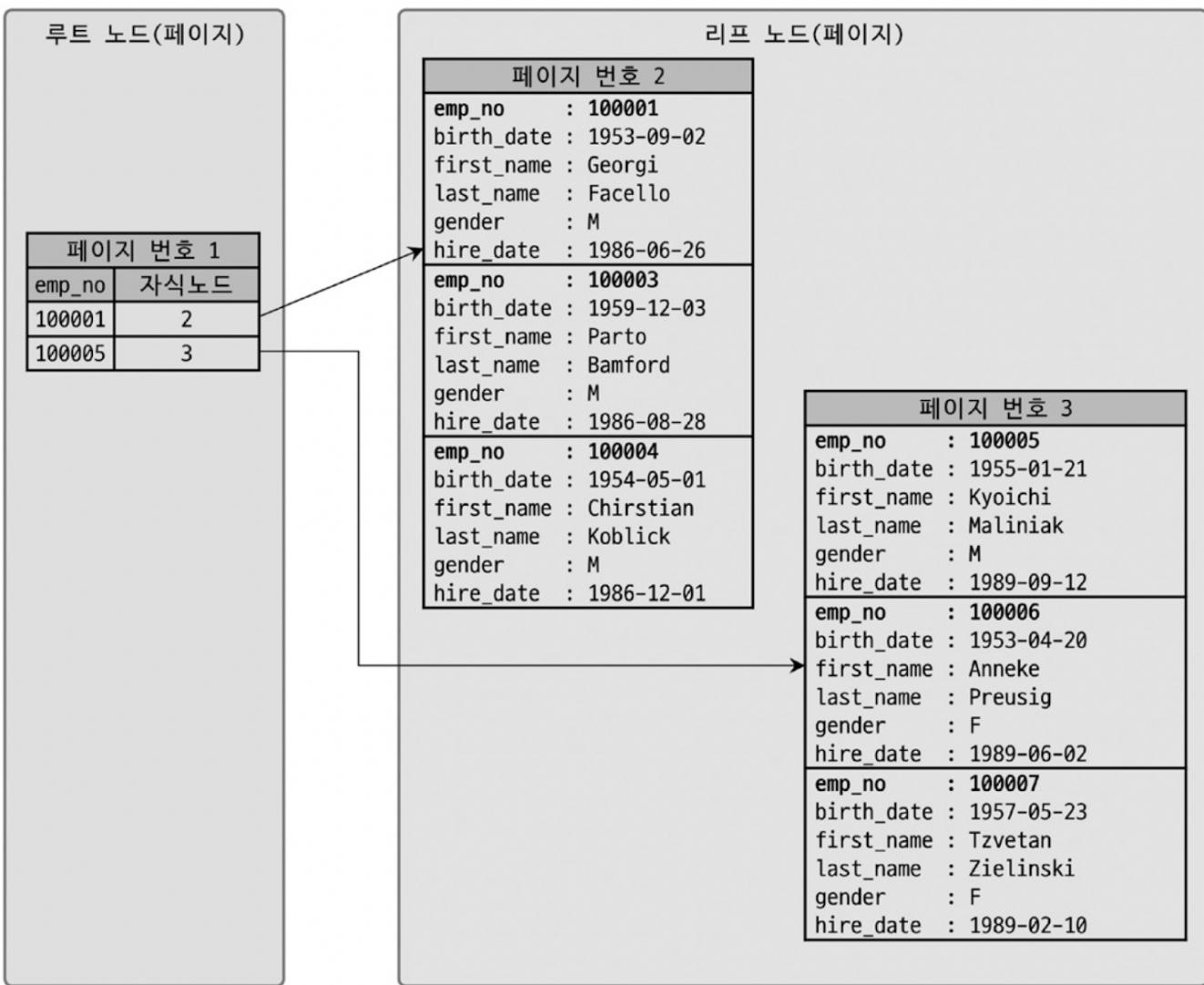


그림 8.25 클러스터링 테이블(인덱스) 구조

위 그림에서 클러스터링 인덱스 구조를 보면 클러스터링 테이블의 구조 자체는 B-Tree와 비슷합니다.

다만 세컨더리 인덱스를 위한 B-Tree의 리프 노드와 달리 클러스터링 인덱스의 리프 노드에는 레코드의 모든 컬럼이 같이 저장돼 있습니다.

즉, 클러스터링 테이블은 그 자체가 하나의 거대한 인덱스 구조로 관리되는 것입니다.

아래와 같이 클러스터링 테이블에서 PK를 변경하면 다음과 같은 변화가 발생합니다.

```
update tb_test set emp_no =100002 where emp_no =100007;
```

위 그림에서 emp\_no가 100007인 레코드는 3번 페이지에 저장돼 있습니다. 하지만 100002로 변경되면서 2번 페이지로 이동하게 되는데 (100002가 없음)

실제로 pk를 변경하는 것은 권장하지 않음

만약 프라이머리 키가 없는 InnoDB 테이블은 다음과 같은 방법으로 클러스터링 테이블을 구성하는 방법은 다음과 같습니다.

먼저 프라이머리 키가 없는 경우 InnoDB 스토리지 엔진이 다음 우선순위대로 프라이머리 키를 대체할 컬럼을 선택합니다.

1. 프라이머리 키가 있으면 기본적으로 프라이머리 키를 클러스터링 키로 선택
2. Not Null 옵션의 유니크 인덱스 중에서 첫 번째 인덱스를 클러스터링 키로 선택
3. 자동으로 유니크한 값을 가지도록 증가되는 컬럼을 내부적으로 추가한 후, 클러스터링 키로 선택

InnoDB 스토리지 엔진이 적절한 클러스터링 키 후보를 찾지 못하는 경우 InnoDB 스토리지 엔진이 내부적으로 레코드의 일련번호 컬럼을 생성합니다.

이렇게 자동으로 추가된 프라이머리 키는 사용자에게 노출되지 않으며, 쿼리 문장에 명시적으로 사용할 수 없습니다.

즉 프라이머리 키나 유니크 인덱스가 전혀 없는 InnoDB 테이블에서 클러스터링 인덱스는 테이블당 단 하나만 가질 수 있기에 꼭 몇시적으로 프라이머리 키를 사용하는 것을 권장합니다.

## 세컨더리 인덱스에 미치는 영향

InnoDB 스토리지 엔진에서 세컨더리 인덱스가 실제 레코드가 저장된 주소를 가지고 있다면 다음과 같은 문제가 있습니다.

먼저 클러스터링 키 값이 변경될 때마다 데이터 레코드의 주소가 변경되고 그때마다 해당 테이블의 모든 인덱스에 저장된 주솟값을 변경해야 합니다.

이런 오버헤드를 제거하기 위해 InnoDB 테이블(클러스터링 테이블)의 모든 세컨더리 인덱스는 해당 레코드가 저장된 주소가 아니라 프라이머리 키값을 저장하도록 구현돼 있다.

이렇게 하면 클러스터링 키 값이 변경될 때마다 모든 인덱스의 주소를 변경할 필요가 없으므로 성능 및 오버헤드를 최적화할 수 있습니다.

## 클러스터링 인덱스의 장점과 단점

클러스터링되지 않은 일반 프라이머리 키와 클러스터링을 비교하면 다음과 같은 장단점이 있습니다.

### 장점

- 프라이머리 키로 검색할 때 처리성능이 매우 빠름
  - 특히 범위 검색
- 테이블의 모든 세컨더리 인덱스가 프라이머리 키를 가지고 있기 때문에 인덱스만으로 처리될 수 있는 경우가 많은
  - 커버링 인덱스라고 이야기함

### 단점

- 테이블의 모든 세컨더리 인덱스가 클러스터링 키를 갖기 때문에 클러스터링 키 값의 크기가 클 경우 전체적으로 인덱스의 크기가 커짐
- 세컨더리 인덱스를 통해 검색할 때 프라이머리 키로 다시 한번 검색해야 하므로 처리 성능이 느림 (이게 뭐였지?)

### 1. 세컨더리 인덱스의 동작 원리:

- 세컨더리 인덱스는 주로 테이블의 특정 열(column)에 대한 빠른 검색을 제공합니다. 이 인덱스는 해당 열의 값을 키로 하며, 각 값과 해당 값이 위치한 데이터 레코드의 프라이머리 키를 맵핑합니다.
- 세컨더리 인덱스를 사용하여 검색하면 먼저 해당 인덱스를 통해 원하는 값을 찾습니다. 그런 다음 프라이머리 키 값을 얻어냅니다.

### 2. 프라이머리 키로 재검색하는 이유:

- 세컨더리 인덱스는 검색할 열의 값을 키로 가지고 있으므로, 해당 값을 기반으로 레코드의 프라이머리 키를 찾아야 합니다.
- 프라이머리 키는 데이터베이스 내에서 행을 고유하게 식별하는 데 사용되기 때문에, 실제 데이터를 찾기 위해 프라이머리 키로 재검색하는 것이 필요합니다.

### 3. 성능 저하의 이유:

- 세컨더리 인덱스를 사용한 후 프라이머리 키로 재검색하면 추가적인 I/O 작업이 필요합니다. 인덱스를 검색하는 데 I/O가 발생하고, 이후 프라이머리 키로 검색하면 다시 I/O가 발생하므로 처리 성능이 저하됩니다.
- 또한, 이중 검색으로 인해 CPU 및 메모리 리소스도 더 많이 소비될 수 있습니다.
- INSERT 할 때 프라이머리 키에 의해 레코드의 저장 위치가 결정되기 때문에 처리 성능이 느림
- 프라이머리 키를 변경할 때 레코드를 DELETE하고 INSERT 하는 작업이 필요하기 때문에 처리 성능이 느림 (거의 하지 않는 작업이라 단점이라고 말하기 애매한듯?)

요약하면 클러스터링 인덱스의 장점은 빠른 읽기이며 단점은 느린쓰기입니다.

일반적으로 온라인 트랜잭션 환경에서는 읽기가 높은 비율을 차지하기 때문에 빠른 읽기를 빠르게 유지하는 것은 매우 중요합니다.

## 클러스터링 테이블 사용시 주의 사항

### 클러스터링 인덱스 키의 크기

클러스터링 테이블의 경우 모든 세컨더리 인덱스가 프라이머리 키값을 포함합니다.

그래서 프라이머리 키의 크기가 커지면 세컨더리 인덱스도 자동으로 크기가 커집니다.

하지만 일반적으로 테이블에 세컨더리 인덱스가 4~5개 정도 생성된다는 것을 고려하면 세컨더리 인덱스 크기는 급격히 증가합니다.

5개의 세컨더리 인덱스를 가지는 테이블의 프라이머리 키가 10바이트인 경우와 50바이트인 경우를 비교하면 다음과 같습니다.

프라이머리 키 크기	레코드당 증가하는 인덱스 크기	100만 건 레코드 저장 시 증가하는 인덱스 크기
10바이트	10바이트 * 5 = 50 바이트	50바이트 * 1,000,000 = 47MB
50바이트	50바이트 * 5 = 250 바이트	250바이트 * 1,000,000 = 238MB

레코드 한 건 한 건을 생각하면 50바이트는 크지 않지만 레코드 건수가 100만 건만 돼도 인덱스의 크기가 거의 190MB나 증가했고 1000만건이 되면 1.9GB가 증가합니다.

또한 인덱스가 커질수록 같은 성능을 내기 위해 그만큼의 메모리가 더 필요해지므로 InnoDB 테이블의 프라이머리 키는 신중하게 선택해야 합니다.

## 프라이머리 키는 Auto-INCREMENT 보다는 업무적인 컬럼으로 생성

InnoDB의 프라이머리 키는 클러스터링 키로 사용되며 이 값에 의해 레코드의 위치가 결정됩니다.

즉 프라이머리 키로 검색하는 경우 클러스터링되지 않은 테이블에 비해 매우 빠르게 처리될 수 있음을 의미합니다.

프라이머리 키는 중요한 역할을 하기 때문에 검색에서 상당히 빈번하게 사용되기 때문에 업무적으로 해당 레코드를 대표할 수 있다면 그 컬럼을 프라이머리키로 설정하는 것이 좋다.

곰곰히 생각해봤는데

a.i가 아닌 다른 방식으로 설정했을 때 예시로 학번이 있지 않을까 합니다.

년도-학과코드-번호 이렇게 구성되는데

이런 용도 때문에 그런 것 같아서 추측합니다.

<https://www.phpschool.com/gnuboard4/bbs/board.php?>

[bo\\_table=qna\\_db&wr\\_id=174609&page=607](#)

## Auto-Increment 컬럼을 인조 식별자로 사용할 경우

여러 개의 컬럼이 복합으로 프라이머리 키가 만들어지는 경우 프라이머리 키의 크기가 길어질 때가 가끔 있습니다.

하지만 프라이머리 키의 크기가 길어도 세컨더리 인덱스가 필요하지 않다면 그대로 프라이머리키를 사용하는 것이 좋습니다.

세컨더리 인덱스도 필요하고 프라이머리 키의 크기도 같다면 Auto\_incrmenet컬럼을 추가하고, 이를 프라이머리 키로 설정하면 됩니다.

이렇게 프라이머리 키를 대체하기 위해 인위적으로 추가된 프라이머리키를 인조 식별자라고 합니다.

그리고 로그 테이블 같이 조회보다는 INSERT 위주의 테이블은 AUTO\_INCRMENET를 이용한 인조 식별자를 프라이머리 키로 설정하는 것이 성능 향상에 도움이 됩니다.

## 유니크 인덱스

- 유니크는 사실 인덱스라기보다는 제약 조건에 가깝습니다.
- 말 그대로 테이블이나 인덱스에 같은 값이 2개 이상 저장될 수 없음을 의미하는데

- MySQL에서는 인덱스 없이 유니크 제약만으로 설정할 방법이 없습니다.
- 유니크 인덱스에서 NULL도 저장될 수 있는데 NULL은 특정 값이 아니므로 2개 이상 저장될 수 있습니다.
- MySQL에서 프라이머리 키는 기본적으로 NULL을 허용하지 않는 유니크 속성이 자동으로 부여됩니다.

## 유니크 인덱스와 일반 세컨더리 인덱스의 비교

유니크 인덱스와 유니크하지 않은 일반 세컨더리 인덱스는 인덱스의 구조상 아무런 차이점이 없습니다.  
단 읽기와 쓰기 성능 관점에서 바라보면 차이가 있습니다.

### 인덱스 읽기

유니크 인덱스와 일반 세컨더리 인덱스는 성능차이가 거의 없습니다.

### 인덱스 쓰기

유니크 인덱스의 경우 인덱스 쓰기는 일반 세컨더리 인덱스와 비교했을 때 중복된 값이 있는지 없는지 체크하는 과정이 있기 때문에 세컨더리 인덱스의 쓰기보다 느립니다.

심지어 중복된 값을 체크할 때 읽기 잠금을 사용하고 쓰기를 할 때 쓰기 잠금이 발생하는데 이 과정에서 데드락이 빈번하게 발생한다.

또한 인덱스 키의 저장을 버퍼링하기 위해 체인지 버퍼가 사용되는데 이는 인덱스의 저장이나 변경 작업이 상당히 빨리 처리되지만 중복 체크 작업으로 인해 일반 세컨더리 인덱스보다 변경 작업이 느리게 작동한다.

## 유니크 인덱스 사용시 주의사항

꼭 필요한 경우라면 유니크 인덱스를 생성하는 것은 당연합니다.

하지만 더 성능이 좋아질 것으로 생각하고 불필요하게 유니크 인덱스를 생성하는 것은 좋지 않습니다.

1. 하나의 테이블에서 같은 컬럼에 유니크 인덱스와 일반 인덱스를 각각 중복해서 생성해 둔 경우가 가끔 있는데 MySQL의 유니크 인덱스는 일반 다른 인덱스와 같은 역할을 하므로 중복해서 인덱스를 생성할 필요가 없습니다.
2. 똑같은 컬럼에 프라이머리 키와 유니크 인덱스를 동일하게 생성하는 경우도 있는데 불필요한 중복입니다.

## 외래키

- MySQL에서 외래키는 InnoDB 스토리지 엔진에서만 생성할 수 있으며, 외래키 제약이 설정되면 자동으로 연관되는 테이블의 컬럼에 인덱스까지 생성됩니다.
- 외래키가 제거되지 않은 상태에서 자동으로 생성된 인덱스를 삭제할 수 없습니다

InnoDB의 외래키 관리에는 중요한 두 가지 특징이 있습니다.

1. 테이블의 변경(쓰기 잠금)이 발생하는 경우에만 잠금 경합(잠금 대기)이 발생합니다.
2. 외래키와 연관되지 않은 컬럼의 변경은 최대한 잠금 경합을 발생시키지 않습니다.

```

8 |   create table tb_parent(
9 |     id int not null primary key,
10|      fd varchar(100) not null
11|  );
12|
13|
14|  create table tb_child (
15|    id int not null,
16|    pid int default null,
17|    fd varchar(100) default null,
18|    primary key(id),
19|    key ix_parentid (pid),
20|    constraint child_ibfk_1 foreign key (pid)references tb_parent (id)on delete cascade
21|  );
22|
23| ✓  insert into tb_parent value (1, 'parent-1'), (2, 'parent-2');
24| ✓  insert into tb_child value (100, 1, 'child-100') ;

```

## 자식 테이블의 변경이 대기하는 경우

작업 번호	커넥션-1	커넥션-2
1	BEGIN;	
2	UPDATE tb_parent SET fd='changed-2' WHERE id=2;	
3		BEGIN;
4		UPDATE tb_child SET pid=2 WHERE id=100;
5	ROLLBACK;	
6		Query OK, 1 row affected (3.04 sec)

위 작업에서는 1번 커넥션에서 먼저 트랜잭션을 시작하고 부모 테이블에서 id가 2인 레코드에 update를 실행 합니다.

이 과정에서 1번 커넥션이 tb\_parent 테이블에서 id가 2인 레코드에 대해 쓰기 잠금을 획득합니다.

그리고 2번 커넥션에서 자식 테이블의 외래키 컬럼인 pid를 2로 변경하는 쿼리를 실행 해당 쿼리는 부모 테이블의 변경 작업이 완료될 때까지 대기합니다.

다시 1번 커넥션에서 ROLLBACK이나 COMMIT으로 트랜잭션을 종료하면 2번 커넥션의 대기 중이던 작업이 즉시 완료됩니다.

즉 자식 테이블의 외래 키 컬럼의 변경은 부모 테이블의 확인이 필요한데, 이 상태에서 부모 테이블의 해당 레코드가 쓰기 잠금이 걸려있으면 해당 쓰기 잠금이 해제될 때까지 기다리게 됩니다.

자식 테이블의 외래키가 아닌 컬럼의 변경은 외래키로 인한 잠금 확장은 발생하지 않습니다.

## 부모 테이블의 변경 작업이 대기하는 경우

작업 번호	커넥션-1	커넥션-2
1	BEGIN;	
2	UPDATE tb_child SET fd='changed-100' WHERE id=100;	
3		BEGIN;
4		DELETE FROM tb_parent WHERE id=1;
5	ROLLBACK;	
6		Query OK, 1 row affected (6.09 sec)

첫 번째 커넥션에서 부모 키 "1"을 참조하는 자식 테이블의 레코드를 변경하면 tb\_child 테이블은 레코드에 대한 쓰지 잠금을 획득합니다.

이 상태에서 2번 커넥션이 tb\_parent 테이블에서 id가 1인 레코드를 삭제하는 경우 이 쿼리는 tb\_child 테이블의 레코드에 대한 쓰기 잠금이 해제될 때 까지 기다려야 한다.

이는 자식 테이블이 생성될 때 정의한 외래키의 특성 (on delete cascade) 때문에 부모 레코드가 삭제되면 자식 레코드도 동시에 삭제되는 식으로 작동하기 때문이다.

데이터베이스에서 외래 키를 물리적으로 생성하려면 이러한 현상으로 인한 잠금 경합까지 고려해 모델링 하는 것이 좋습니다.