5.3 B-Tree 인덱스

생성일	@2021년 6월 13일 오후 5:23
태그	

B-Tree에서 B는 바이너리가 아니라 Balanced를 의미한다

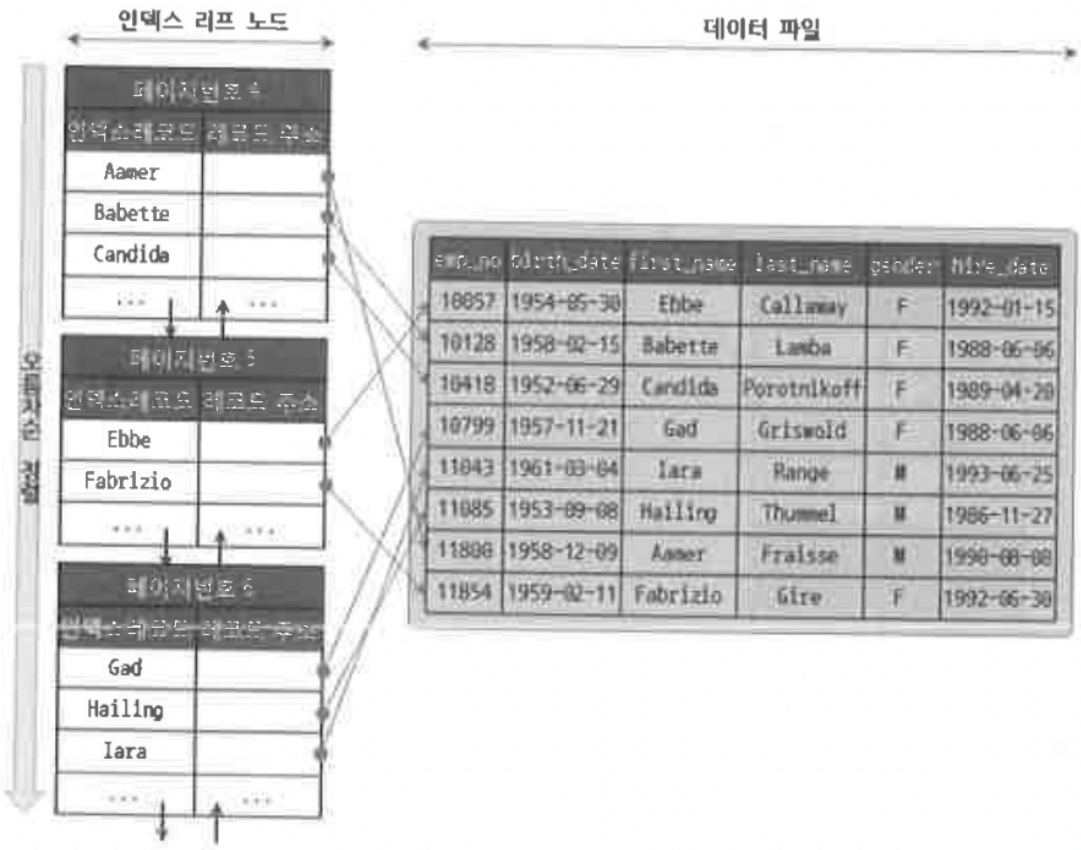
DBMS에서는 주로 B+ Tree또는 B* Tree가 사용됨

1. 구조 및 특성

- 루트 노드 : 트리 구조 최상단에 존재
- 자식 노드 : 루트 노드 하단에 붙어있는 노드
- 리프 노드 : 트리구조의 최 하단에 있는 노드. ***실제 데이터 레코드를 찾아가기 위한 주소값을 가지고 있음***
- 브랜치 노드: 트리구조에서 루트도 아니고 리프노드도 아닌 중간 노드를 일컬음

인덱스의 키값은 모두 정렬되어있지만, 데이터 파일의 레코드는 정렬돼 있지 않고 임의의 순서대로 저장되어있음. 레코드가 삭제되어 빈 공간이 생기면 그 다음의 데이터 INSERT는 삭제된 공간을 재활용

InnoDB 테이블에서 레코드는 클러스터(비슷한 값들을 최대한 모아서 저장하는 방식)되어 디스크에 저장되어서, 기본적으로 프라이머리 키 순서대로 정렬되어 저장됨



[그림 5-7] B-Tree의 리프 노드와 테이블 데이터 레코드

인덱스의 리프노드는 데이터 파일에 저장된 레코드의 주소를 가지게 되는데, 스토리지 엔진에 따라 `레코드 주소` 의미가 달라짐

- 오라클 : 물리적인 레코드 주소
- InnoDB : 프라이머리 키 값

2. B-Tree 인덱스 키 추가 및 삭제

인덱스 키 추가

새로운 키값이 B-Tree에 저장될 때 테이블의 스토리지 엔진에 따라 새로운 키값이 즉시 인덱스에 저장될 수도 있고 그렇지 않을 수도 있음

1. B-Tree에 새로 키값이 저장되는 경우 : 먼저 트리 상 적절한 위치 검색
2. 대상 레코드의 주소 정보를 B-Tree 리프노드에 저장

3. 만약 리프노드가 꽉차서 더 저장 불가능할때 리프노드는 Split
4. 이후 상위 브랜치 노드까지 처리

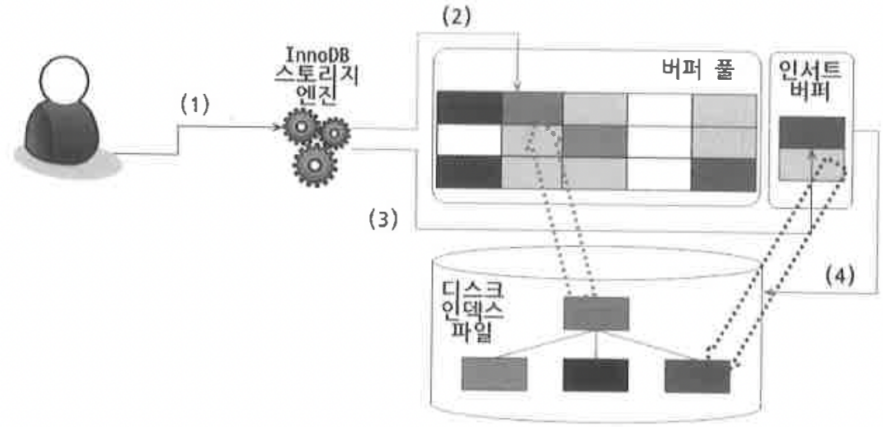
→ WRITE작업에 비용이 많이 드는 편

테이블에 레코드를 추가하는 작업 비용을 1이라고 가정

- 해당 테이블의 인덱스에 키를 추가하는 작업비용 : 1~1.5
- 테이블에 인덱스가 3개가 있다면 (1.5 *3 + 1) 5.5 정도로 예측 가능

인서트 버퍼 (InnoDB)

상황에 따라 인덱스 키 추가 작업을 지연 시켜 나중에 처리할지, 아니면 바로 처리할지 결정



[그림 5-8] 인서트 버퍼의 처리

(1) 사용자의 쿼리 실행

(2) InnoDB의 버퍼 풀에 새로운 키값을 추가해야 할 페이지(B-Tree의 리프 노드)가 존재한다면 즉시 키 추가 작업 처리

(3) 버퍼 풀에 B-Tree의 리프 노드가 없다면 인서트 버퍼에 추가할 키값과 레코드의 주소를 임시로 기록해 두고 작업 완료(사용자의 쿼리는 실행 완료됨)

(4) 백그라운드 작업으로 인덱스 페이지를 읽을 때마다 인서트 버퍼에 머지해야 할 인덱스 키값이 있는지 확인한 후, 있다면 병합함(B-Tree에 인덱스 키와 주소를 저장).

(5) 데이터베이스 서버 자원의 여유가 생기면 MySQL 서버의 인서트 버퍼 머지 스레드가 조금씩 인서트 버퍼에 임시 저장된 인덱스 키와 주소 값을 머지(B-Tree에 인덱스 키와 주소를 저장)시킴

MySQL 5.5 이상 버전부터...

- INSERT 뿐만 아니라 DELETE 등에 의한 인덱스 키의 추가 및 삭제 작업까지 버퍼링해서 지연 처리 할 수 있게 기능이 확장됨 (Change Buffering으로 이름이 변경됨)
- innodb_change_buffering 설정 값을 통해 키 추가/삭제 작업 중 어느 것을 지연처리할지 결정
- 인서트 버퍼에 의해 인덱스 키 추가 작업이 지연되어 처리된다 하더라도 사용자에게 영향X

인덱스 키 삭제

B-Tree의 키 값이 삭제되는 경우 very 간단

- 삭제 될 키 값이 저장된 B-Tree 리프노드를 찾아서 삭제 마크 (끝) → 그대로 방치 또는 재활용 가능
- 이 작업 또한 버퍼링되어 지연처리가 될 수 있음

인덱스 키 변경

B-Tree의 키 값이 변경되는 경우, 단순히 인덱스 상의 키값만 변경하는것 불가능

먼저 키값을 삭제한 후 다시 새로운 키값을 추가하는 형태로 처리됨

인덱스 키 검색

Tree Traversal : B-Tree의 루트노드부터 시작해서 브랜치노드를 거쳐 리프노드 까지 이동하며 비교작업 수행

- SELECT 뿐만 아니라 UPDATE나 DELETE를 처리하기 위해 항상 해당 레코드를 먼저 검색해야할 경우에도 인덱스가 있으면 빠른 검색이 가능
- B-Tree 인덱스를 이용한 검색은 100%일치 또는 값의 앞부분만 일치하는 경우에 사용 가능
  - <> 나 뒷부분 일치는 B-Tree 인덱스 검색 불가능
- 인덱스의 키 값에 변형이 가해진 뒤 비교되는 경우, 절대 B-Tree의 빠른 검색 기능을 사용할 수 없다

- ex) 함수나 연산을 수행한 결과로 정렬 / 검색하는 작업

> 💡 InnoDB 스토리지 엔진에서 인덱스
> : 레코드 잠금이나 넥스트 키 락 (갭 락)이 검색을 수행한 인덱스를 잠근 후 테이블의 레코드를 잠그는 방식으로 구현이 되어있다
>
> → UPDATE, DELETE 문장이 실행될때, 테이블에 적절히 사용할 수 있는 인덱스가 없으면 불필요하게 많은 레코드를 잠금

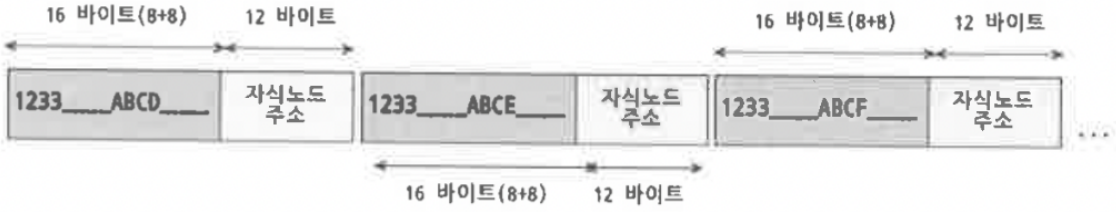
3. B-Tree 인덱스 사용에 영향을 미치는 요소

인덱스 키 값의 크기

- 페이지(Page) 또는 블록

  : InnoDB 스토리지 엔진이 디스크에 데이터를 저장하는 가장 기본 단위

  - 디스크의 모든 읽기/쓰기 작업의 최소 단위
  - 페이지는 InnoDB 스토리지 엔진의 버퍼 풀에서 데이터를 버퍼링하는 기본 단위이기도 함
  - 인덱스도 페이지 단위로 관리됨 (루트/브랜치/리프 노드 구분한 기준이 Page단위)



[그림 5-9] 인덱스 페이지의 구성

💡 Q : MySQL의 B-Tree는 자식 노드를 몇 개까지 가질까?
A : 인덱스의 페이지 크기와 키값의 크기에 따라 결정됨

ex) InnoDB의 모든 페이지 크기는 16KB로 고정돼 있음 (+ 자식 노드 주소가 평균 12바이트로 구성된다고 가정)
16*1024/(16+12) = 585개 저장 가능

→ 키값의 크기가 만약 32바이트로 늘어났다? 그럼 16*1024/(32+12) = 372개 저장 가능
이렇게 되면 SELECT 쿼리가 레코드 500개 이상을 읽어야 할때 후자의 경우 두 번 읽어야함.. (느려진다는 뜻)

B-Tree 깊이

B-Tree 인덱스의 깊이(Depth)는 상당히 중요하지만 직접적으로 제어할 방법이 없다.

ex) 위 예제의 연장선으로 B-Tree의 깊이가 3인 경우 최대 몇개의 키값을 가질 수 있나?

키 값이 16바이트인경우, 최대 2억(585*585*585)

키 값이 32바이트인경우, 5천만 (372 * 372*372)

값을 검색할 때 몇번이나 랜덤하게 디스크를 읽어야 하는지와 직결되는 문제

결론적으로 인덱스 키값의 크기가 커지면 커질 수록 하나의 인덱스 페이지가 담을 수 있는 인덱스 키값의 개수가 작아짐 → 같은 레코드 건수여도 B-Tree의 깊이가 깊어져서 디스크 읽기가 더 많이 필요하게 된다

💡 인덱스 키값의 크기는 작으면 작을 수록 좋다..

선택도 (기수성)

인덱스에서 선택도(Selectivity) 또는 기수성(Cardinality)는 거의 같은 의미

= 인덱스 키 값 가운데 유니크한 값의 수를 의미

- 인덱스 키값 가운데 중복된 값이 많아지면 많아질 수록 카디널리티가 낮아지고, 그럼 속도도 느려짐

읽어야하는 레코드의 건수

테이블에 레코드가 100만건이 있을때...

전체 테이블을 모두 읽은 뒤 필요없는 50만 버리기 vs 인덱스를 통해 필요한 50만건만 읽어오기

→ 둘 중 어느게 효율적일지 판단해야함 (전자가 효율적임. 어차피 옵티마이저가 알아서 처리함)

옵티마이저에서는 인덱스를 통해 레코드를 1건 읽는것이 테이블에서 직접 레코드 1건 읽는 것보다 4~5배 정도 더 비용이 많이 드는 작업

→ 인덱스를 통해 읽어야할 레코드의 건수가 전체 테이블 레코드의 20~25% 이상이면 직접 테이블을 모두 읽어서 필터링 방식으로 처리하는 것이 효율적

4. B-Tree 인덱스를 통한 데이터 읽기

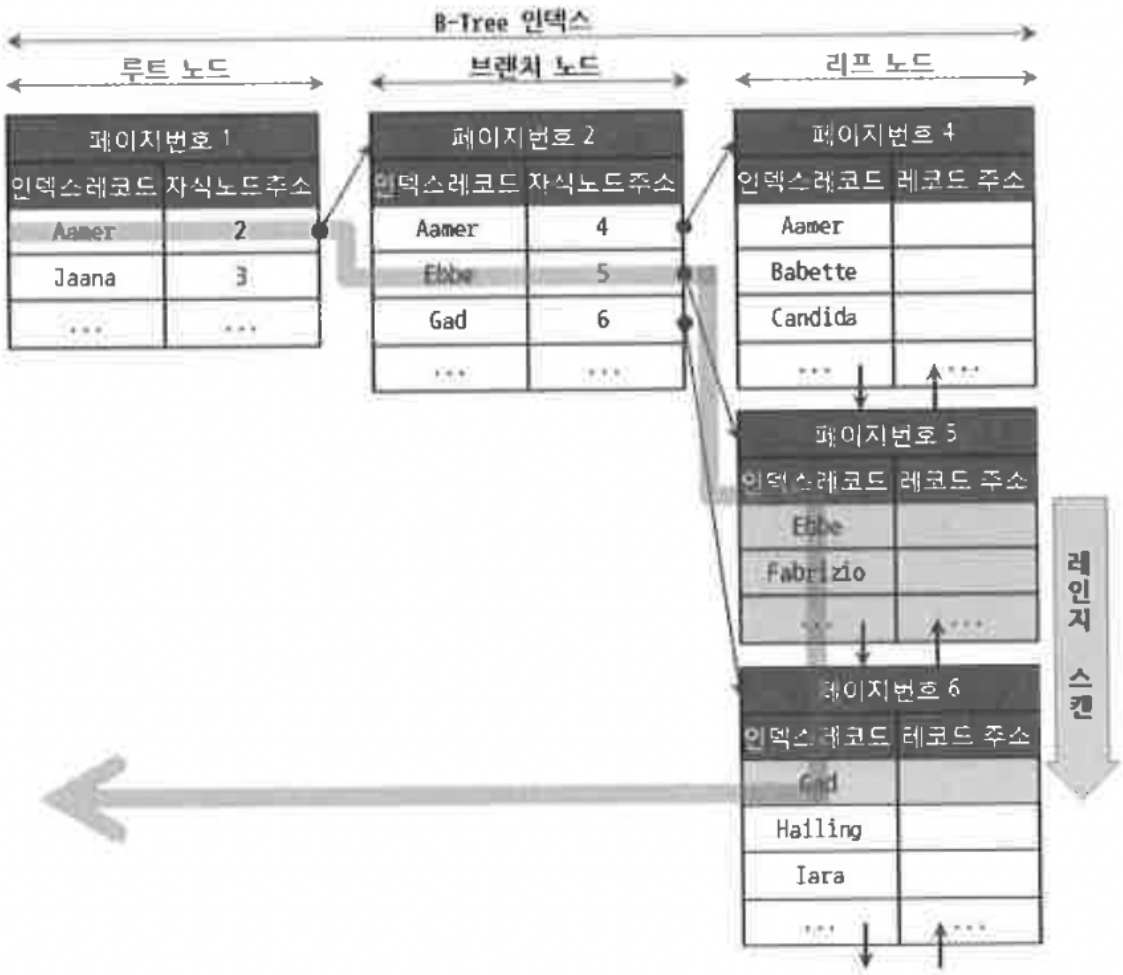
어떤 경우에 인덱스를 사용하도록 유도할지, 또는 사용하지 못하게 할지 판단하려면 어떻게 해야함?

인덱스를 이용하는 대표적인 방법 3가지

1. 인덱스 레인지 스캔

- 가장 대표적인 접근 방식
- 나머지 2개 방식보다 빠른 방법

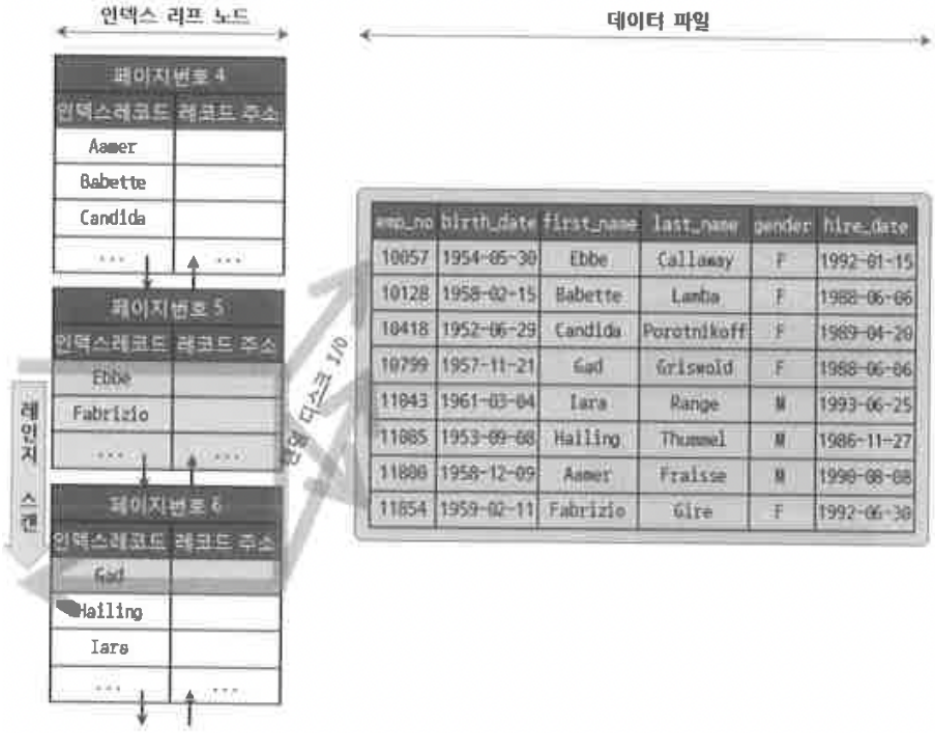
```
SELECT * FROM employees WHERE first_name BETWEEN 'Ebbe' AND 'Gad';
```



[그림 5-9] 인덱스를 이용한 레인지 스캔

인덱스 레인지 스캔: 검색해야 할 인덱스의 범위가 결정되었을 때 사용하는 방식

- 루트 노드부터 비교 시작 → 브랜치 노드 거쳐서 → 리프노드까지 찾아서 들어가야 원하는 시작지점 찾을 수 있음
- 시작 지점을 찾고나서는 리프노드의 레코드만 순서대로 읽으면됨
- 스캔 중 리프 노드의 끝까지 읽으면, 리프 노드 간의 링크를 이용해 다음 리프 노드를 찾아서 다시 스캔
- 최종적으로 스캔을 멈춰야 하는 위치에 다다랐을때 사용자에게 지금까지 읽은 레코드를 반환



[그림 5-10] 인덱스 레인지 스캔을 통한 데이터 레코드 읽기

- B-Tree 인덱스에서 루트와 브랜치 노드를 이용해 특정 검색 시작 값을 가지고 있는 리프 노드를 검색.
- 그 지점부터 필요한 방향으로 인덱스를 읽어나감 (오름차순/내림차순)
- 리프 노드에 저장된 레코드 주소로 데이터 파일의 레코드를 읽어옴
  - 이때 레코드 한 건 단위로 랜덤 I/O가 한번씩 실행됨
  - 3건의 레코드가 일치했다고 가정하면 데이터 레코드를 읽기 위해 랜덤 I/O가 최대 3번이 필요한것
  - 그래서 인덱스를 통해 데이터 레코드를 읽는 작업은 비용이 많이 드는 작업으로 분류되는 것..

2. 인덱스 풀 스캔

인덱스 레인지 스캔과 달리 인덱스의 처음부터 끝까지 모두 읽는 방식

- 쿼리의 조건절에 사용된 칼럼이 인덱스의 첫번째 칼럼이 아닌경우
- 인덱스는 (A, B, C) 칼럼 순서대로 만들어져 있지만 쿼리의 조건절은 B나 C칼럼으로 검색하는 경우

테이블 풀 스캔하는것보다 인덱스만 읽는 것이 효율적임

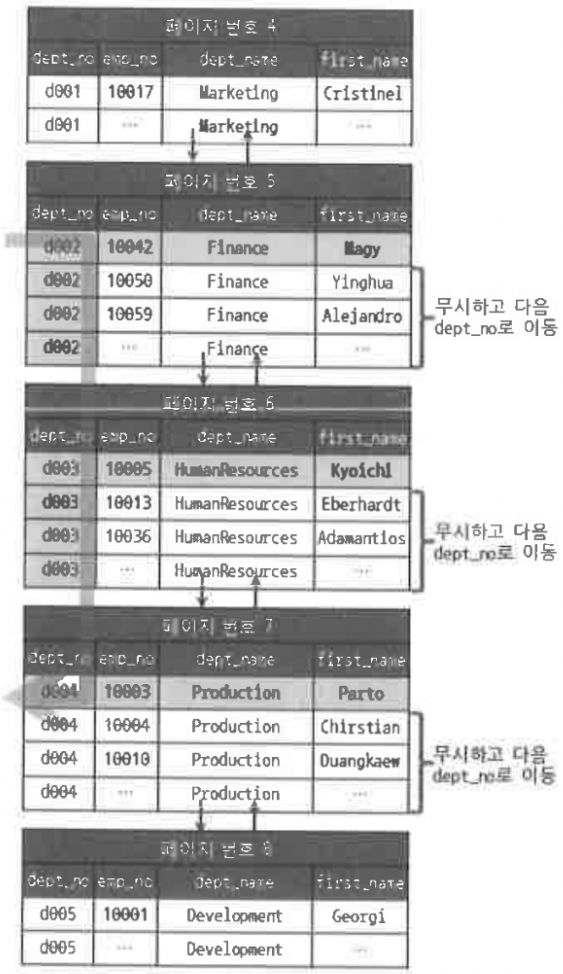
→ 쿼리가 인덱스에 명시된 칼럼만으로 조건을 처리할 수 있는 경우 주로 이 방식이 사용됨

→ 인덱스뿐만 아니라 데이터 레코드까지 모두 읽어야한다? 그럼 절대 이방식으로 처리 안됨

3. 루즈 인덱스 스캔

느슨하게,, 듬성듬성하게 인덱스를 읽는 것

```
SELECT dept_no, MIN(emp_no)
FROM dept_emp
WHERE dep_no BETWEEN 'd002' AND 'd004'
GROUP BY dept_no;
```



[그림 5-12] 루스 인덱스 스캔(dept_name과 first_name 칼럼은 참조용으로 표시됨)

인덱스 레인지 스캔과 비슷하게 작동하지만, 중간마다 불필요한 인덱스 키값은 SKIP하고 다음으로 넘어가는 형태

- GROUP BY 또는 집합 함수(MAX, MIN)에 대해 최적화를 하는 경우에 사용됨