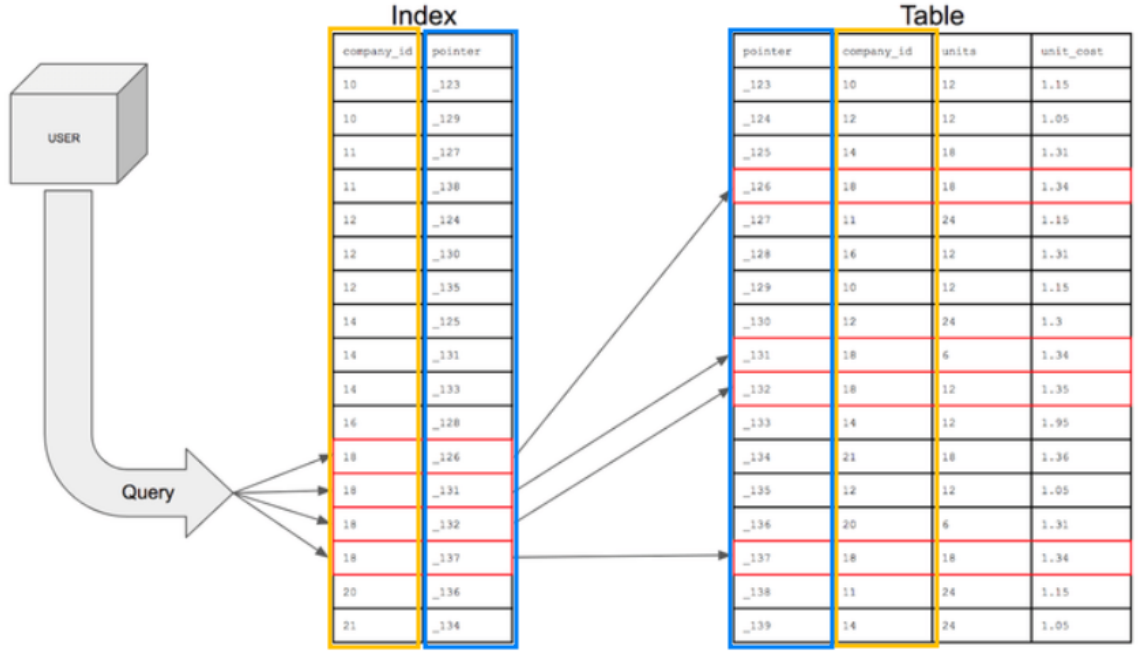
[DB] Index

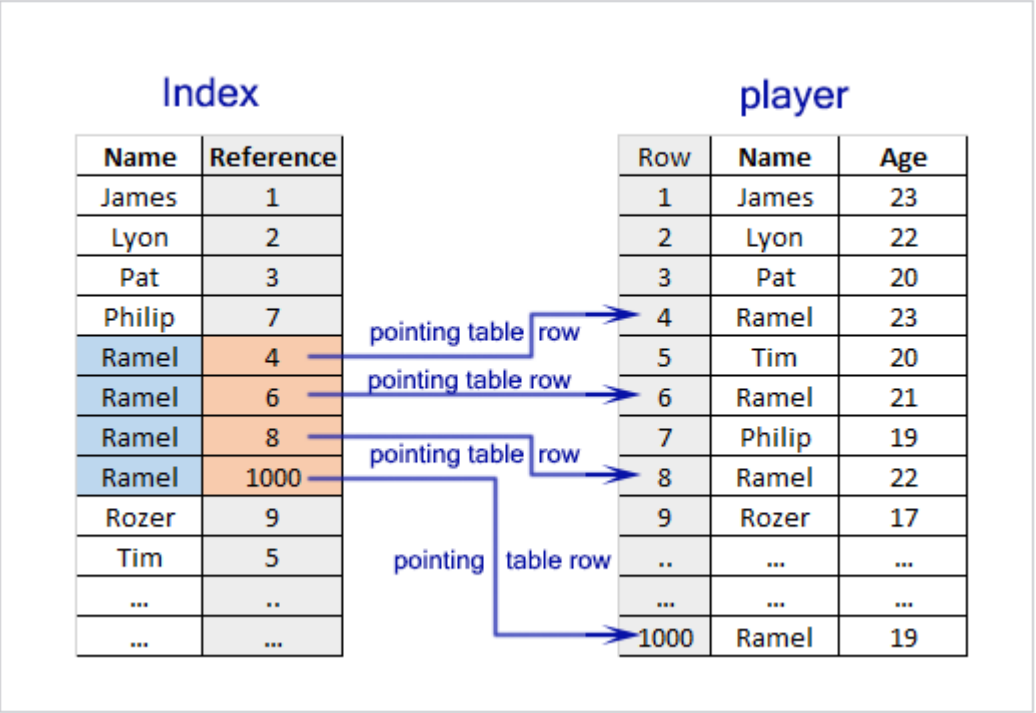
인덱스

- 인덱스는 DBMS의 검색 속도를 향상시키기 위해 만들어진 자료 구조이다. 테이블의 모든 data를 full scanning하면 시간이 오래 걸리기 때문에 데이터와 데이터의 위치를 포함한 자료구조를 생성하여 빠르게 조회할 수 있도록 도와준다.
- 아래는 테이블에서 company_id와 pointer만 따로 빼서 company_id를 기준으로 오름차순으로 정렬하여 index를 만든 모습이다.



ex) company_id = 18인 컬럼을 검색할 때, 그냥 테이블에서 검색하면 처음부터 끝까지 company_id를 확인해서 18인 data를 찾는다.
하지만 인덱스를 사용하면 18인 데이터를 빠르게 찾을 수 있다!

▼ 사진이 잘 안보여서.. 다른 예



- 내부적으로 칼럼의 값과 물리적 주소를 Key, Value의 트리 형태로 저장한다.
- 책에 비유한 인덱스 개념
  - 보통 책의 맨 끝에 존재하는 색인 페이지 → DBMS의 인덱스
  - 책의 내용 → DBMS의 데이터 파일
    - 데이터 파일: DB의 데이터를 담는 파일로, 모든 DB는 하나 이상의 데이터 파일을 갖는다.
  - 책의 색인을 통해 알 수 있는 페이지 번호 → DBMS의 데이터 파일에 저장된 레코드의 주소

  추가로, 책의 색인은 사전 순으로 정렬되는데 DBMS의 인덱스도 일정 기준으로 정렬 가능하다!

장점

- 테이블에서 검색, 정렬 속도를 향상시킨다.
  - 데이터들이 정렬되어있기 때문에 조건에 맞는 데이터를 빠른 속도로 찾을 수 있다.
  - ORDER BY, MIN, MAX의 경우 데이터를 정렬하는 데 부하가 많이 걸리기 때문에 이미 정렬되어있는 인덱스를 사용하면 효율적인 처리가 가능하다.

단점

- 인덱스를 항상 최신의 상태로 정렬시켜야 하기 때문에 인덱스가 적용된 칼럼에 INSERT, DELETE, UPDATE가 수행될 때 마다 추가적인 연산이 필요하며, 그에 따른 오버헤드가 발생한다.
    - DELETE, UPDATE는 기존 인덱스를 삭제하는 것이 아니라 '사용하지 않음' 처리를 하고 남겨둔다. 따라서, 수정 작업이 많은 경우 실제 데이터에 비해 인덱스가 과도하게 커지는 문제가 발생하고, 성능이 오히려 저하될 수 있다.
- 인덱스를 저장하기 위한 추가 저장 공간(DB의 약 10%)이 필요하다.

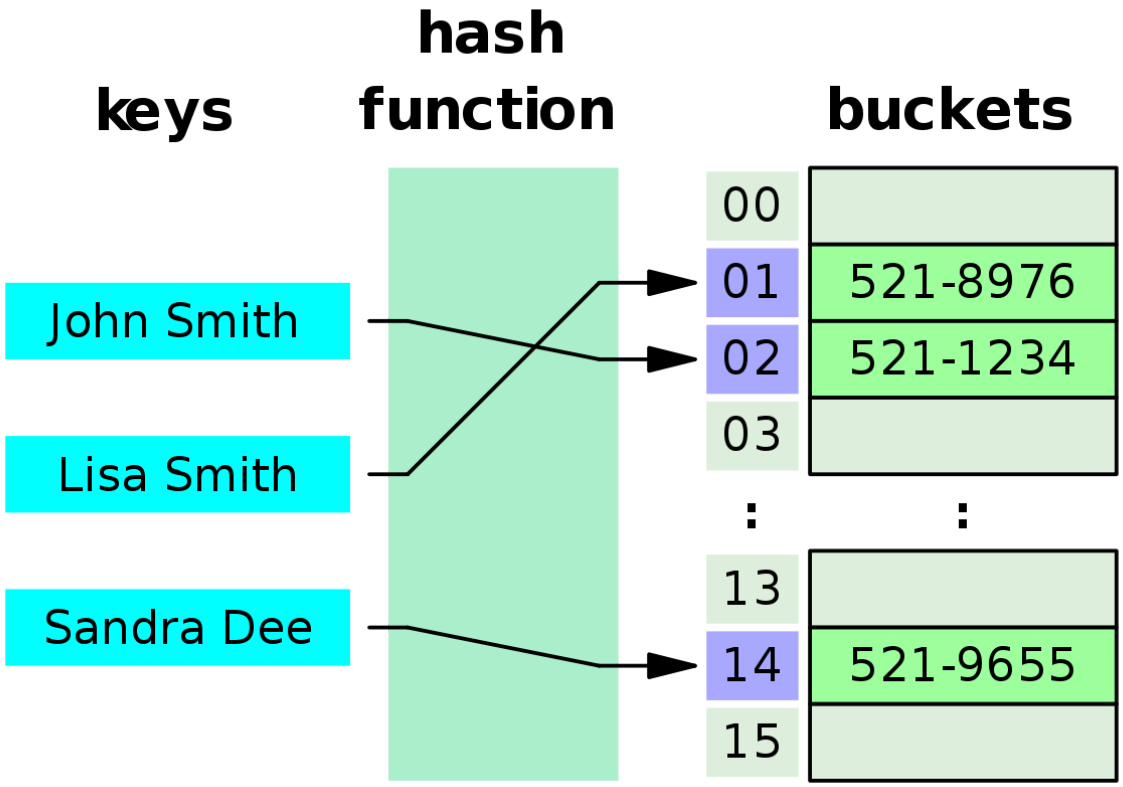
인덱스를 사용하면 좋은 예

- 규모가 큰 테이블
- INSERT, UPDATE, DELETE가 자주 발생하지 않는 컬럼
- SELECT 쿼리의 WHERE, ORDER BY, JOIN이 자주 사용되는 컬럼
- 중복된 데이터가 적은 컬럼 (Cardinality가 높은 컬럼)
    - 인덱스는 key-value 형태로 데이터를 저장하기 때문에 중복된 key가 여러개 있다면 검색할 대상이 많아진다.
    - 인덱스로 많은 부분을 걸러낼수록 효율이 높아진다. 만약 성별을 인덱스로 잡는다면, 남 or 여 두 가지이므로 50%밖에 걸러내지 못한다.

Hash Table 인덱스

해시 테이블은 (Key, Value)로 데이터를 저장한다.

해시 테이블 인덱스는 Key값을 이용해 고유한 index를 생성하여 그 index에 저장된 값을 꺼내온다.

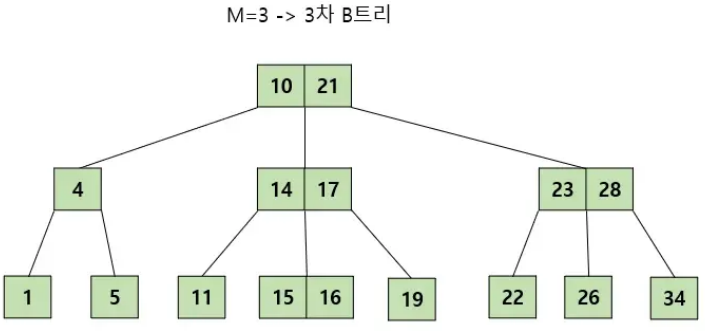


하지만 해시 테이블은 등호(=) 연산에만 특화되어 있고, 값의 크고 작음을 비교하는 ***부등호 연산에는 적합하지 않다.*** 따라서 해시 테이블 인덱스는 매우 제한적으로 사용된다.

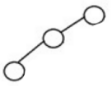
B-Tree 인덱스

대부분의 DBMS는 B-Tree 계열 인덱스를 사용한다.

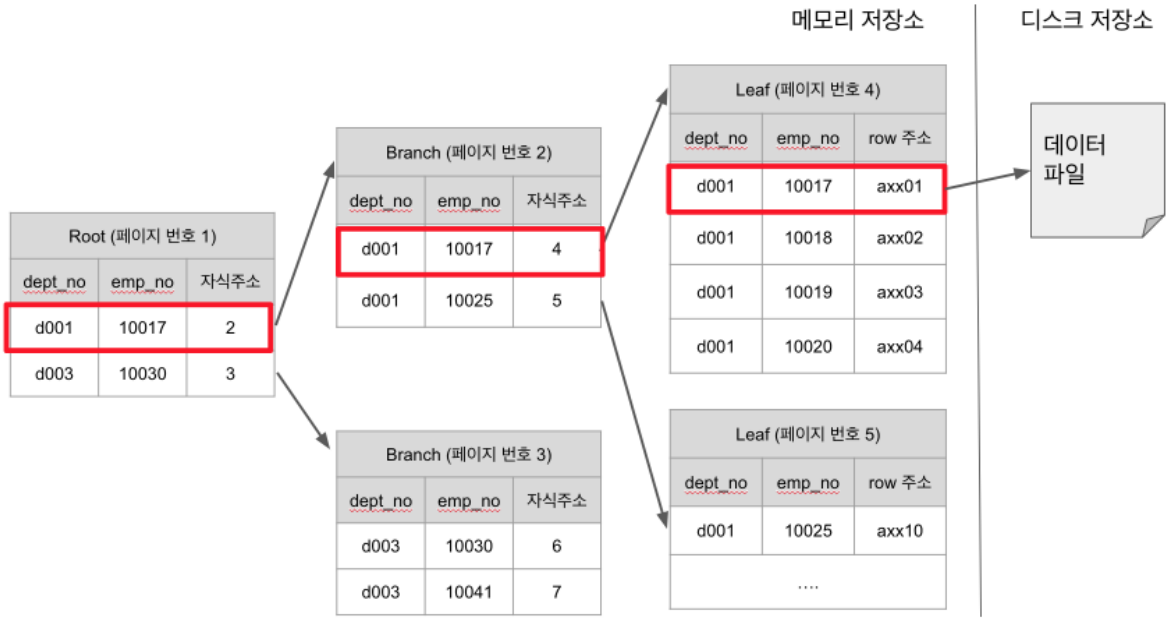
B Tree 특징



- 이진 탐색 트리와 다르게 각 노드가 2개 이상의 자식 노드를 가질 수 있고, 좌우 레벨이 항상 균형을 유지하는 정렬된 트리이다.
- 이진 탐색 트리의 단점을 개선: 이진 탐색 트리에서 연산을 수행할 때, 시간 복잡도가 최악일 때 $O(N)$이다. (편향 이진 트리(오른쪽 참고)를 생각해보자!) 이를 개선하기 위한 트리 구조 중 하나가 B Tree 이다.



  - B Tree의 시간복잡도는 트리의 높이가 균형적으로 유지되기 때문에 탐색, 저장, 수정, 삭제에도 동일하게 $O(logN)$



- Root, Branch, Leaf
  - Root Node: 분기 값을 저장하고, 하위의 Branch를 가리키는 항목을 포함한다.
  - Branch Node: 분기 값을 저장하고, 하위의 Leaf를 가리키는 항목을 포함한다.
  - Leaf Node: 실제 key 값 + ROWID를 저장한다.
- 인덱스 탐색은 Root -> Branch -> Leaf -> 디스크 저장소 순으로 진행된다.
  - Branch (페이지번호 2) 는 dept_no가 d001이면서 emp_no가 10017 ~ 10024까지인 Leaf의 부모로 있다.
  - `dept_no=d001 and emp_no=10018` 라고 조회하면 페이지 번호 4인 Leaf를 찾아 데이터 파일의 주소를 불러와 반환한다.

B-Tree 계열 인덱스를 주로 사용하는 이유

- B-Tree는 위처럼 노드 하나에 여러 데이터가 저장될 수 있다. **각 노드 내 데이터들은 항상 정렬된 상태**로 특정 값보다 크고 작음을 비교하는 **부등호 연산**에 문제가 없다.
- 참조 포인터가 적어 데이터 양이 많을 경우에도 빠른 메모리 접근이 가능하다.
- **항상 좌, 우 자식노드 개수의 균형이 맞게 유지**하므로 데이터 탐색뿐 아니라, 저장, 수정, 삭제에도 항상 **O(logN)**의 시간 복잡도를 가진다.