# DB - Index

## 💡 Index

☑️ 데이터베이스 테이블에 대한 검색 성능 속도를 높여주는 자료구조

- 두꺼운 책의 목차와 같음
- SELECT query 실행 시, 정렬되어 있는 index에서 빠른 속도로 검색 가능
- 일반적인 RDBMS에선 `B+Tree` 구조로된 index를 사용하여 검색속도 향상

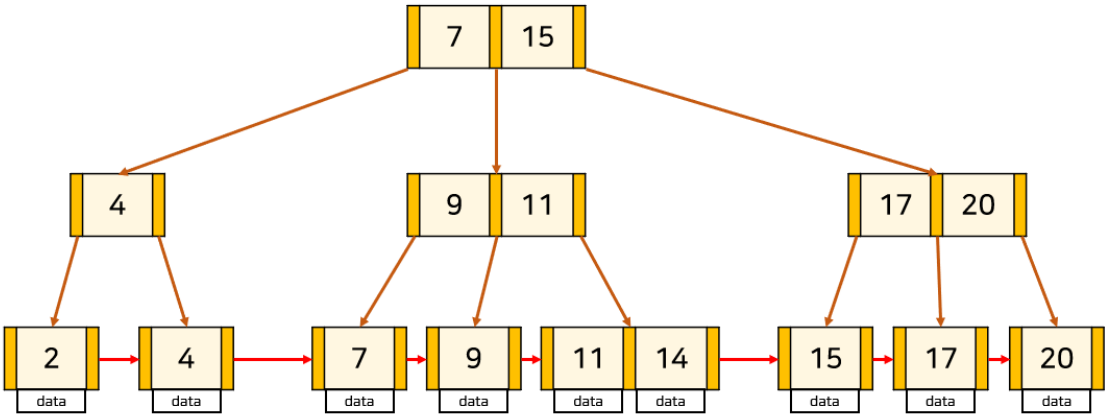
## 💡 Index 구조

- Btree, B+tree, Hash, Bitmap
- 특정 컬럼에 인덱스를 생성하면, 해당 컬럼의 `데이터들을 정렬`하여 별도의 메모리 공간에 `데이터의 물리적 공간`과 함께 함께 저장
- `search-key`: Index에 저장되는 속성 값
- `pointer`: 실제 데이터의 물리적 위치를 저장한 값
- index는 순서대로 정렬된 search-key값과 pointer값만 저장하기 때문에 table보다 적은 공간 차지

📌 특정 column을 search-key 값으로 설정하여 index 생성
→ 해당 search-key값을 기준으로 정렬하여 (search-key, pointer)를 별도 파일에 저장

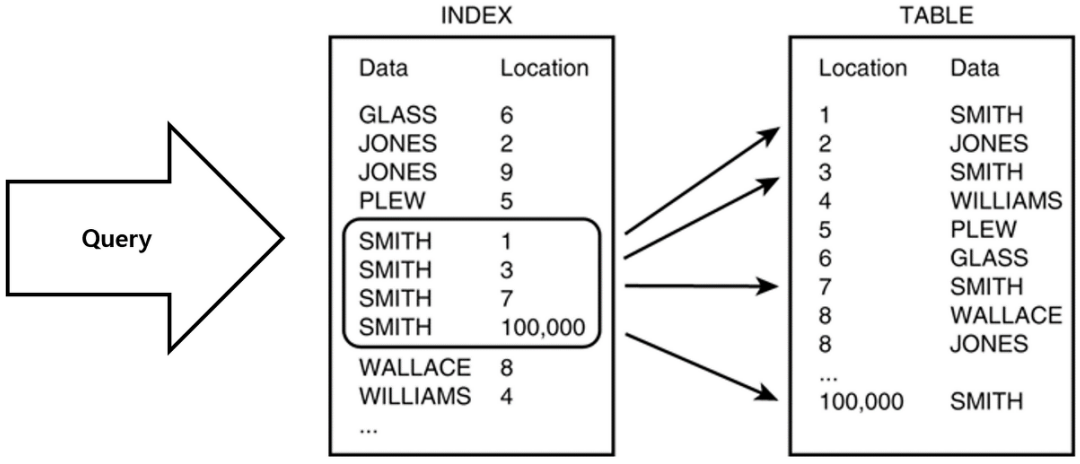
## 💡 Index 관리 방식

- `B+Tree 자료구조`




- 이진 탐색 트리와 유사
- Root Node(기준) / Branch Node(중간) / Leaf Node(말단)으로 구성되는 계층적 구조
- Node는 데이터가 존재하는 공간
- Leaf Node만 인덱스(Key)와 함께 데이터(Value)를 가지고 있고, 나머지 Root Node와 Branch Node는 데이터를 위한 인덱스(Key)만을 가짐
- Leaf Node에만 데이터를 저장하고 Leaf Node들끼리 Linked List로 연결되어 있어 선형 시간이 소모되어 시간 효율 증가
- Root Node에서 경로를 확인 후, 그에 알맞는 Node들로 이동하여 최종적으로 원하는 데이터가 있는 Leaf Note에 도달

- `HashTable 자료구조`




- `컬럼의 값`과 `물리적 주소`를 `(key, value)`의 한 쌍으로 저장하는 자료구조

- 등호(=) 연산에 최적화되어 있기 때문에 실제로 Index에서 자주 사용 X
- 데이터베이스에선 부등호(<, >) 연산이 자주 사용되는데, 해시 테이블 내의 데이터들은 정렬되어 있지 않으므로 부등호 연산에 대한 시간 효율성이 낮음

## 💡 Index 파일 구성

- 테이블 생성 시, 3가지 파일 생성
  - `FRM`: 데이블 구조 저장 파일
  - `MYD`: 실제 데이터 파일
  - `MYI`: Index 정보 파일 (Index 사용 시 생성)
- 사용자가 쿼리를 통해 Index를 사용하는 컬럼을 검색하면, `MYI` 파일의 내용을 활용

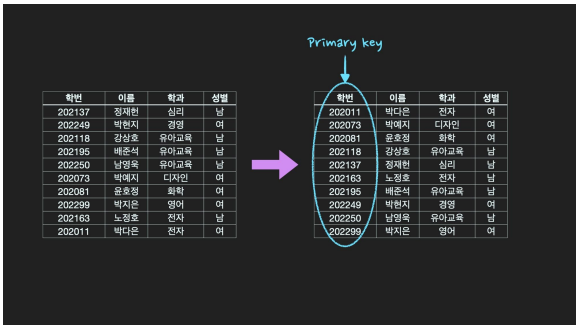
## 💡 Index 사용 이유

- 내부적으로 데이터는 Table에 순서 없이 저장됨
  - Table의 row(record)를 처음부터 끝까지 모두 접근하여 검색조건과 비교하는 `Full Table Scan` 실행
- 하지만 특정 컬럼에 대한 Index를 생성해 놓은 경우
  - 해당 속성에 대하여 search-key가 정렬되어 저장 → 조건 검색(SELECT ~WHERE) 속도 향상

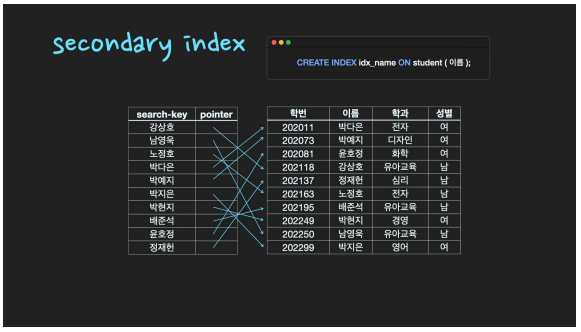
## 💡 클러스터형 인덱스 & 보조 인덱스 (clustering index & secondary index)

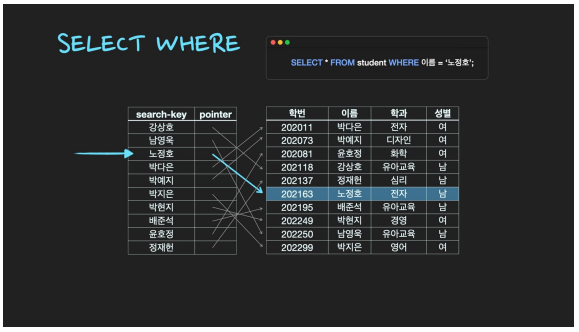
- **clustering index**
  - 특정 컬럼을 `기본키(primary key)`로 지정하면 자동으로 클러스터형 인덱스를 생성하고, 해당 컬럼 기준으로 정렬
  - Table 자체가 정렬된 하나의 index
  - ex) 영어사전
- **secondary index**

- 별도의 공간에 index가 생성
- `create index`와 같은 index 생성 또는 `고유키(unique key)`로 지정하면 보조 인덱스 생성
- ex) 일반 책의 찾아보기








# 💡Index의 장단점

## ✔️장점

- 조건 검색(SELECT ~ WHERE) 속도 향상
  - Full Table Scan을 실행하지 않고, 정렬된 Index로 빠르게 검색 가능
- 정렬 (Order by) 정의 효율성
  - Index를 통해 Order by에 의한 Sort 과정 생략 가능
- MIN, MAX 절의 효율적인 처리
  - Index로 정렬된 데이터에서 MIN, MAX를 효율적으로 추출

## ✔️단점

- 추가 저장공간 필요

    - Index 자료구조를 위한 저장 공간이 추가적으로 필요 (.mdb 파일의 크기 증가)
    - 보통 table 크기의 10%정도의 공간 차지
- 데이터 변경 작업이 느림
    - INSERT, UPDATE, DELETE가 자주 발생하면 성능 감소
    - `B+tree` 구조의 Index는 데이터가 추가 삭제될 때마다 tree의 구조가 변경될 수 있기 때문
    - 즉, `index의 재구성`이 필요하므로 추가적인 자원 소모
- 한 페이지를 동시에 수정할 수 있는 병행성 감소

## 💡 Index를 사용하면 좋은 경우

1. `Where 절`에서 자주 사용하는 Column
2. `외래키(Foriegn key)`가 사용되는 Column
3. `Join`에 자주 사용되는 Column

### 판단 기준

- 카디널리티가 높은가?
    - `카디널리티` : 값의 균형을 나타내는 개념
        - 카디널리티가 가장 높은 필드는 모든 레코드에 다른 값이 들어가있는 유일 키 필드
        - 반대로 모든 레코드에 같은 값이 들어가 있다면 카디널리티가 낮은 필드
    - 값이 평균치에서 많이 흩어져있을수록 좋은 인덱스 후보
- 선택률이 낮은가?
    - `선택률` : 특정 필드값을 지정했을 때 테이블 전체에서 몇 개의 레코드가 선택되는지 나타내는 개념
        - 예를 들어 100개의 레코드를 가진 테이블에서 유일키로 'pkey=1'처럼 등호를 지정한다면 한 개의 레코드가 선택됨. 즉, 선택률은 1/100 = 0.01로 1%임
    - 한 번의 선택으로 레코드가 조금만 선택되는 것이 좋은 후보
    - 선택률이 5% 미만이라면 인덱스를 작성할 가치가 있다고 판단
    - 선택률이 20%보다 높다면 Table Full Scan을 하는 것이 더 빠를 가능성

## 💡 Index 사용을 피해야 하는 경우

1. Data 중복도가 높은 Column
2. DML(INSERT, DELETE, UPDATE)가 자주 일어나는 Column