인덱스

▼ 인덱스

- 데이터베이스 테이블에 대한 **검색 성능을 향상**시키는 자료 구조이며 **WHERE** 절 등을 통해 활용
- 특정 조건을 만족하는 데이터를 찾을 때, full table scan하지 않고(**Range scan**) **B+ Tree**로 구성된 구조에서 **Index 파일 검색**으로 속도를 향상시키는 기술

특징

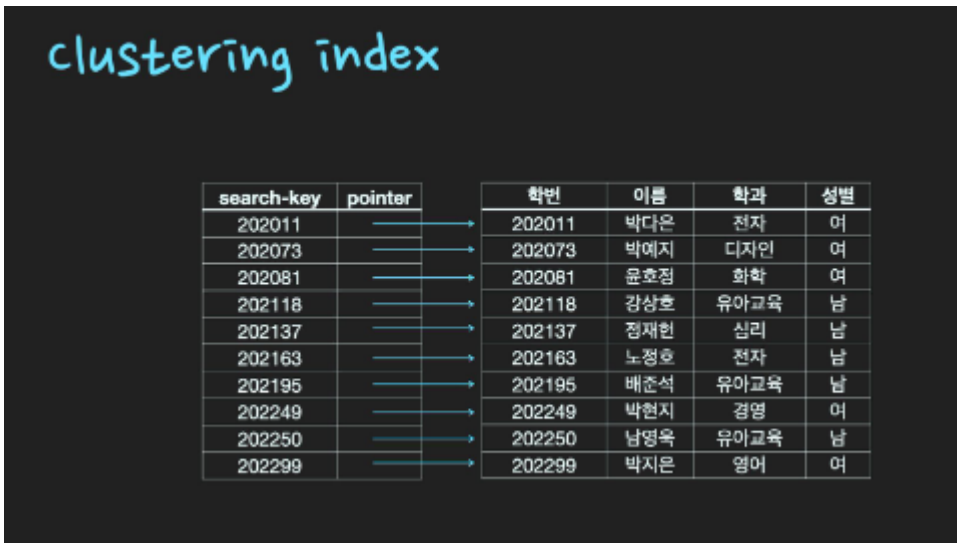
- 항상 **최신의 정렬상태**를 유지
- 인덱스도 하나의 데이터베이스 객체
- 데이터베이스 크기의 약 **10% 정도의 저장공간 필요**
- 인덱스는 **메모리에 저장**
- **Range scan**: 조건을 **찾으면 탐색 중단**
- 대부분의 RDBMS에는 **primary key 생성 시 index가 자동 생성**됨
- **optimizer가 알아서** 적절하게 index를 선택 (풀스캔쓸지 인덱스 쓸지는 옵티마이저가)
- 테이블 당 4~5개 정도 권장
    - 한 테이블에 index가 너무 많으면 데이터 수정 시 소요되는 시간이 너무 길어짐

사용 이유

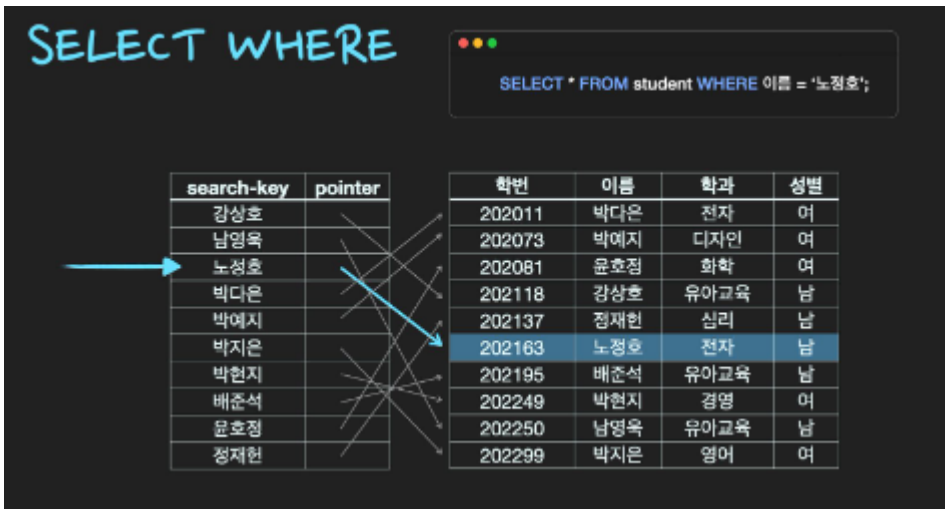
- full scan을 피하기 위해
- **조건을 만족**하는 튜플을 **빠르게 조회**하기 위해
- **빠르게 정렬**(order by)하거나 **그룹핑**(group by) 하기 위해

클러스터형 인덱스와 보조 인덱스

- **클러스터형 인덱스 (Clustering index)**
    - 특정 컬럼을 **기본키(primary key)**로 지정하면 자동으로 클러스터형 인덱스가 생성되고, 해당 컬럼 기준으로 정렬
    - **Table 자체가** 정렬된 하나의 **index**인 것
    - 테이블당 **1개**만 존재 가능



- **보조 인덱스 (Secondary index)**
    - **별도의 공간**에 인덱스가 생성
    - `create index`와 같이 index를 생성하기를 하거나 **고유키(unique key)**로 지정하면 보조 인덱스 생성
    - 테이블당 **여러 개** 존재



장점

- **검색 속도 향상 (SELECT~WHERE~)**
  - 테이블의 RECORD는 내부적으로 순서가 없이 뒤죽박죽으로 저장됨
  - 이렇게 되면 WHERE절을 통해 특정 조건에 맞는 데이터들을 찾아낼 때에도 Full table scan 해야함
  - index에는 데이터들이 정렬되어 저장되어 있기 때문에 검색 조건에 일치하는 데이터들을 빠르게 찾아낼 수 있음

단점

- **추가적인 저장 공간 필요**
  - table의 크기의 10%정도
- **느린 데이터 변경 작업**
  - 데이터 변경, 즉 INSERT, UPDATE, DELETE가 자주 발생하면 성능이 나빠질 수 있음
  - B+ Tree구조의 index는 데이터가 추가 삭제 될 때마다 트리의 구조가 변경되어 인덱스 재구성이 필요하기 때문에 추가적인 자원 소모
- **DELETE, UPDATE**는 기존 인덱스를 삭제하는 것이 아니라 '사용하지 않음' 처리를 하고 남겨둠. 따라서, 수정 작업이 많은 경우 **실제 데이터에 비해 인덱스가 과도하게 커지는 문제**가 발생하고, 성능이 오히려 저하될 수 있음
- 이미 데이터가 몇백만건 이상 있는 테이블에 인덱스를 생성하는 경우 시간이 몇분 이상 소요될 수 있고 DB 성능에 안좋은 영향을 줄 수 있음

Full scan이 더 좋은 경우

- **Table**에 데이터가 **조금** 있을 때 (몇십, 몇백건 정도)
- **조회하려는 데이터**가 테이블의 **상당 부분**을 차지할때
  - ex) 100만건 중 80만건 이상이 조회하려는 데이터일 때

▼ 인덱스를 사용하면 좋은 컬럼

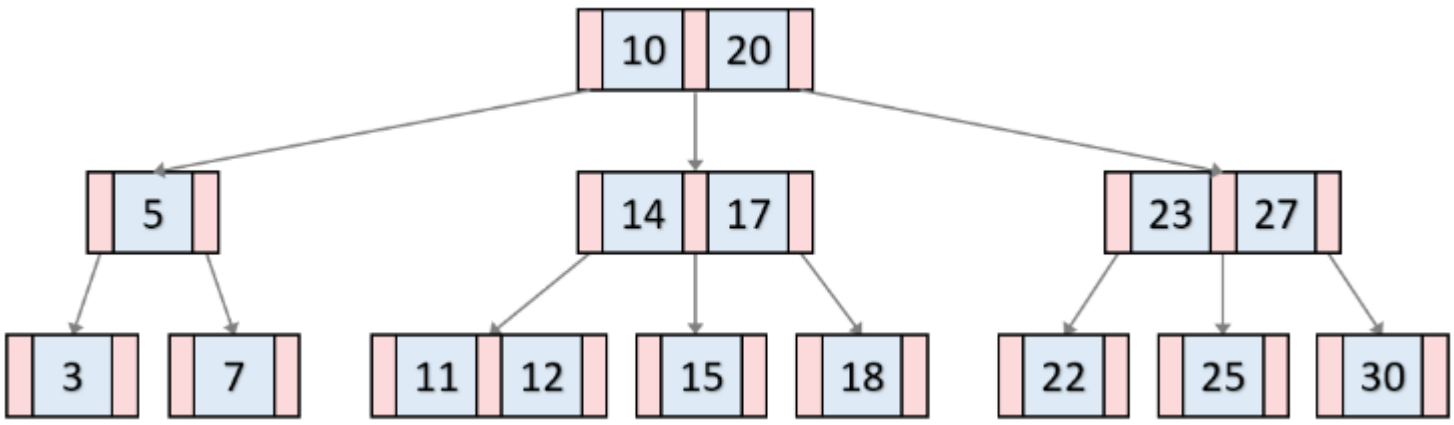
기준	적합성
카디널리티(Cardinality)	높을수록 적합 (데이터 중복이 적을수록 적합)
선택도(Selectivity)	낮을수록 적합
조회 활용도	높을수록 적합 (where 절에서 많이 사용되면 적합)
수정 빈도	낮을수록 적합

- **데이터 중복이 낮은** 컬럼 (=선택도가 낮은 컬럼)
  - 보통 5~10% 이내
- **SELECT WHERE절에 자주 사용**되는 컬럼
- **데이터 수정빈도가 낮은** 컬럼

    - DML(insert, update, delete) 작업 시, 데이터 변화가 생기고 index에서는 매번 정렬을 다시하여 이에 따른 부하 발생
- **Join조건으로 자주 사용**되는 컬럼
- **외래키가 사용**되는 컬럼

▼ 인덱스 구조

B Tree

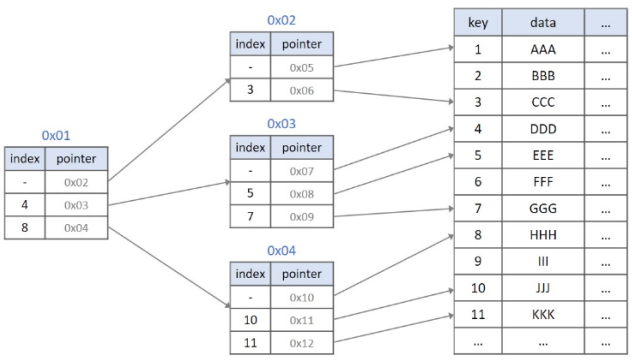


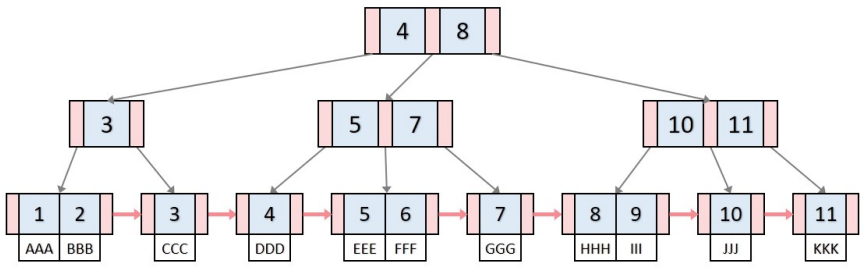
- 이진 탐색트리와 유사한 자료구조
- **자식 노드를 둘이상** 가질 수 있고 **Balanced Tree** 라는 특징이 있다 ⇒ 즉 **탐색 연산에 있어 O(log N)의 시간복잡도**를 갖음
- 모든 노드들에 대해 **값을 저장**하고 있으며 포인터 역할을 동반
- 파란 부분이 각 노드의 key, 빨간 부분이 자식노드를 가르키는 포인터

1. 각 노드에는 2개 이상의 데이터가 저장될 수 있으며, **항상 정렬된 상태**로 저장된다.
    - DB에서 같은 노드 공간의 데이터는 메모리 상 차례대로 저장되어 있어 참조 포인터 값으로 접근하는 것이 아닌, 실제 메모리 디스크에서 바로 다음 인덱스 접근이 가능하다.
2. 각 노드에는 여러 개의 Key를 가지고 있고, 각 Key에 해당하는 데이터와 함께 갖고 있다.
    - Key는 중복되지 않는다.
3. 특정 노드의 **왼쪽** 서브 트리는 해당 노드의 데이터보다 **작은 값**들로, **오른쪽** 서브 트리는 **큰 값**들로 구성되어야 한다.
4. 루트 노드는 적어도 2개 이상의 자식 노드를 가져야 한다.
5. 루트 노드를 제외한 모든 노드는 최소 $M/2$개, 최대 M개의 자식을 가질 수 있다.
    - 최대 M개의 자식을 가질 수 있는 B Tree를 M차 B Tree라 한다.
6. 노드 내 데이터 개수는 $floor(M/2)-1$개부터 최대 $M-1$개까지 포함될 수 있다.
7. 특정 노드의 데이터가 K개라면, 자식 노드의 개수는 $K+1$개여야 한다.
8. **모든 단말 노드는 같은 레벨**에 존재한다.

https://velog.io/@emplam27/자료구조-그림으로-알아보는-B-Tree

B+ Tree





- B-Tree를 개선한 형태의 자료구조
- **값을 리프노드에만** 저장하며 **리프노드는 연결리스트의 형태**를 띄어 선형 검색이 가능

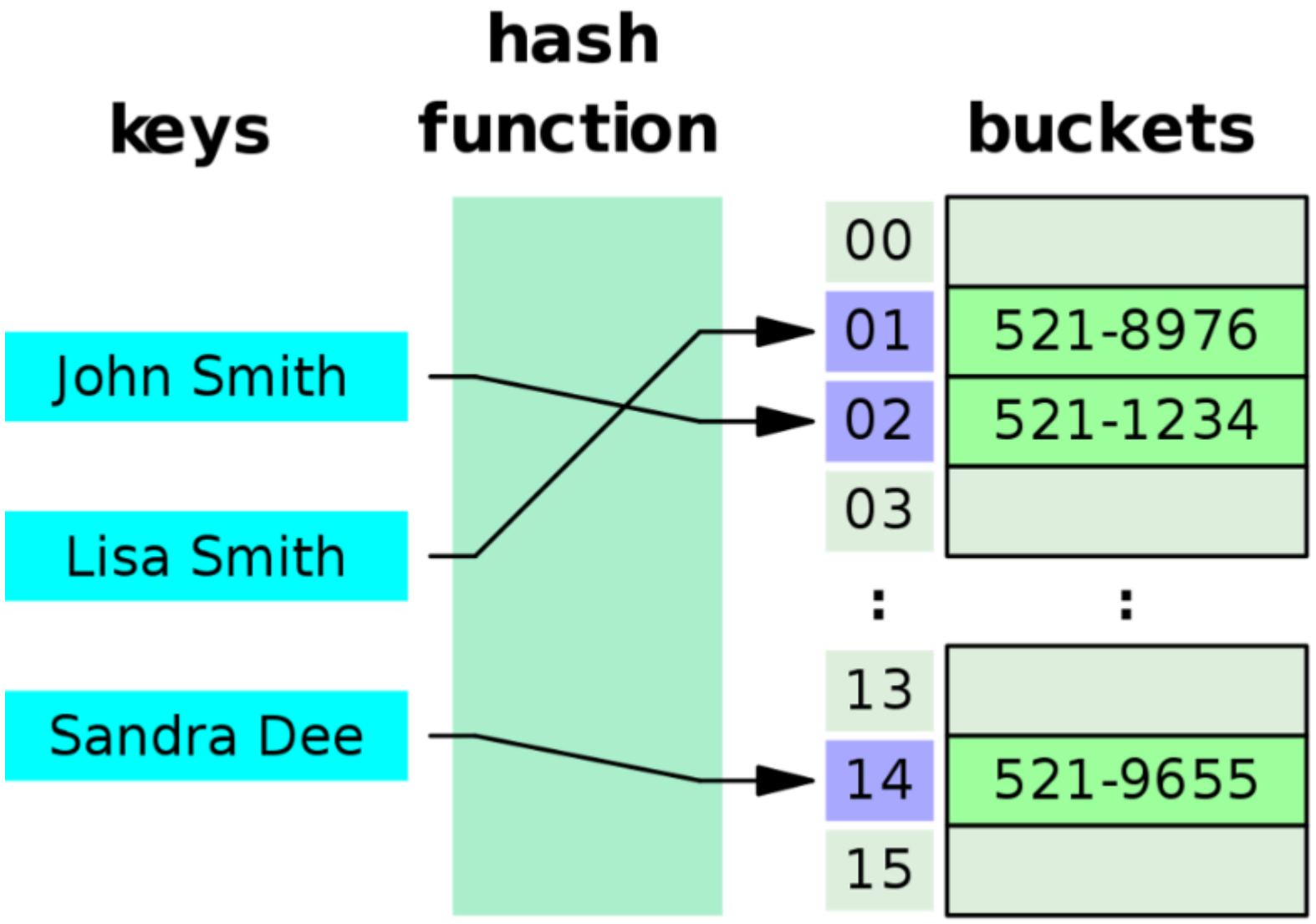
  ⇒ 굉장히 작은 시간복잡도에 검색을 수행할 수 있음

  ⇒ **부등호문 연산에 대해 효과적**
- **리프 노드를 제외**한 노드들은 **포인터의 역할만**을 수행
- **B+트리와 B트리 차이점**
  1. **모든 key, data가 리프노드**에 모여있습니다. B트리는 리프노드가 아닌 각자 key마다 data를 가진다면, B+트리는 리프 노드에 모든 data를 가집니다.
  2. **모든 리프노드가 연결리스트**의 형태를 띄고 있습니다. B트리는 옆에있는 리프노드를 검사할 때, 다시 루트노드부터 검사해야 한다면, B+트리는 리프노드에서 선형검사를 수행할 수 있어 시간복잡도가 굉장히 줄어듭니다.
  3. **리프노드의 부모 key는 리프노드의 첫번째 key보다 작거나 같습니다.** 그림의 B+트리는 리프노드의 key들을 트리가 가지고 있는 경우여서, data 삽입 또는 삭제가 일어날 때 트리의 key에 변경이 일어납니다. 해당 경우뿐만 아니라 data의 삽입과 삭제가 일어날 때 트리의 key에 변경이 일어나지 않게 하여 더욱 편하게 B+트리를 구현하는 방법도 존재하기 때문에 **작거나 같다**라는 표현을 사용하였습니다.

	B-tree	B+tree
주요 특징	모든 내부, 리프 노드들이 데이터를 가진다	단지 리프노드만 데이터를 가진다
검색	모든 키가 리프에서 사용가능 하지 않기 때문에, 검색이 때로 느리다	모든 키가 리프 노드에 있기 때문에 검색이 빠르고 정확하다
중복 키	트리에 중복키가 없다	중복키가 존재하며 모든 데이터들은 리프에 있다
삭제	내부 노드의 삭제는 복잡하고 트리 변형이 많다	어떠한 노드든 리프에 있기 때문에 삭제가 쉽다
리프노드	링크드 리스트로 저장되지 않는다	링크드 리스트로 저장된다
높이	특정 갯수의 노드는 높이가 높다	같은 노드일 때 B-tree보다 높이가 낮다
사용	데이터베이스, 검색엔진	멀티레벨 인덱스, DB 인덱스

https://velog.io/@emplam27/자료구조-그림으로-알아보는-B-Plus-Tree

Hash Table



- 해시 테이블은 **(Key, Value)**로 데이터를 저장
- **해시 함수**를 이용해서 **값을 인덱스로 변경** 하여 관리하는 자료구조
- **일반적**인 경우 탐색, 삽입, 삭제 연산에 대해 **O(1)**의 시간 복잡도

  ⇒ 다른 관리 방식에 비해 빠른 성능을 갖음
- 최악의 경우 **해시 충돌**이 발생하는 것으로 탐색, 삽입, 삭제 연산에 대해 **O(N)**의 시간복잡도를 갖는다.
- 값 자체를 변경하기 때문에 **부등호문, 포함문등의 연산에 사용할 수 없음**

B+tree가 DB index를 위한 자료구조로 적합한 이유

- **항상 정렬된 상태**를 유지하여 **부등호 연산**에 유리
- 데이터 탐색뿐 아니라, **저장, 수정, 삭제**에도 항상 **O(logN)**의 시간 복잡도를 갖음

Hash table의 시간복잡도는 O(1)인데 시간복잡도 O(logN)인 B+tree를 사용하는 이유

- 해쉬 인덱스는 **등호(equality) 비교만 가능하고 범위(range) 비교는 불가능**하기 때문
- **다중인덱스(multicolumn index)의 경우 전체 attributes에 대한 조회만** 가능
  - ex) (a,b) : B트리는 a만 사용하는 경우도 가능한데 해쉬는 불가능