Index

Index

- 추가적인 쓰기 작업과 저장 공간을 활용하여 데이터베이스 테이블의 **검색 속도를 향상**시키기 위한 자료구조
- 테이블의 모든 데이터를 검색하면 시간이 오래 걸리기 때문에 데이터와 데이터의 위치를 포함한 자료구조를 생성하여 빠르게 조회할 수 있도록 도움
- 인덱스를 활용하면, 데이터를 조회하는 **SELECT** 외에도 **UPDATE**나 **DELETE**의 성능이 함께 향상됨 → update, delete 연산을 하기 위해서는 해당 대상을 조회해야 하기 때문
- index를 항상 **최신의 정렬된 상태로 유지**해야 원하는 값을 빠르게 탐색할 수 있음
- INSERT, UPDATE, DELETE가 수행된다면 오버헤드가 발생한다.
    - INSERT: 새로운 데이터에 대한 인덱스를 추가함
    - DELETE: 삭제하는 데이터의 인덱스를 사용하지 않는다는 작업을 진행함
    - UPDATE: 기존의 인덱스를 사용하지 않음 처리하고, 갱신된 데이터에 대해 인덱스를 추가함

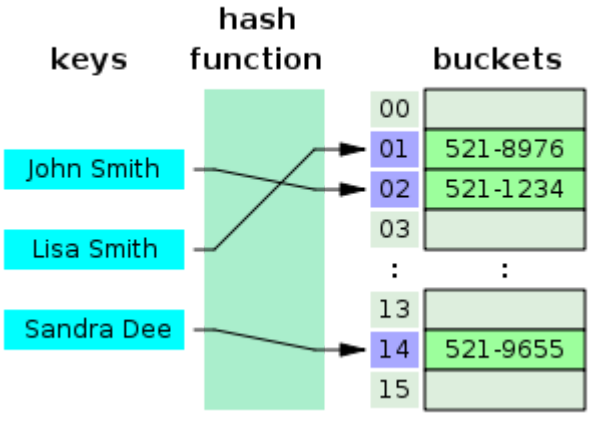
장단점

- **장점**
    - 테이블을 **조회하는 속도와 그에 따른 성능을 향상**시킬 수 있다.
    - 전반적인 **시스템의 부하를 줄일** 수 있다.
- **단점**
    - 인덱스를 관리하기 위해 DB의 약 10%에 해당하는 **저장공간이 필요**하다.
    - 인덱스를 **관리하기 위해 추가 작업**이 필요하다.
    - 인덱스를 잘못 사용할 경우 **오히려 성능이 저하되는 역효과**가 발생할 수 있다.
- **Index를 사용하면 좋은 경우**
    - 규모가 작지 않은 테이블
    - INSERT, UPDATE, DELETE가 자주 발생하지 않는 컬럼
    - JOIN이나 WHERE 또는 ORDER BY에 자주 사용되는 컬럼
    - 데이터의 중복도가 낮은 컬럼 (= cardinality가 높은 컬럼)
- 주의할 점
    - 규모가 작은 테이블 → index의 혜택보단 손해가 더 클 수 있음
    - 한 테이블에 index가 너무 많으면 데이터 수정 시 소요되는 시간이 너무 길어질 수 있음 (테이블 당 4~5개 권장)
    - 사용되지 않는 인덱스는 바로 제거

Index 자료구조

해시테이블

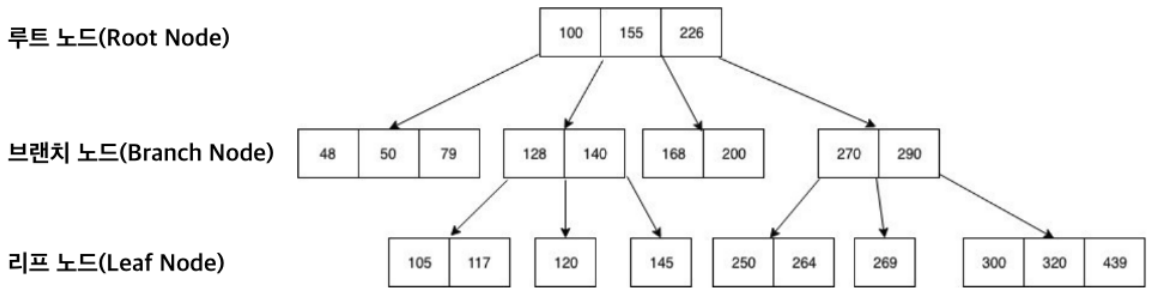
- 빠른 데이터 검색이 필요할 때 유용
- (Key, Value)로 데이터를 저장하는 자료구조
- Key값을 이용해 고유한 index를 생성하여 그 index에 저장된 값을 꺼내오는 구조



- **O(1)의 시간복잡도**로 매우 빠른 검색을 지원
- **해시가 등호(=) 연산에만 특화** → DB 인덱스에서 해시 테이블이 사용되는 경우는 제한적이다
- 해시 함수는 값이 1이라도 달라지면 완전히 다른 해시 값을 생성하기 때문에 **부등호 연산(>, <)이 자주 사용되는 데이터베이스 검색을 위해서는 해시 테이블이 적합하지 않다**.

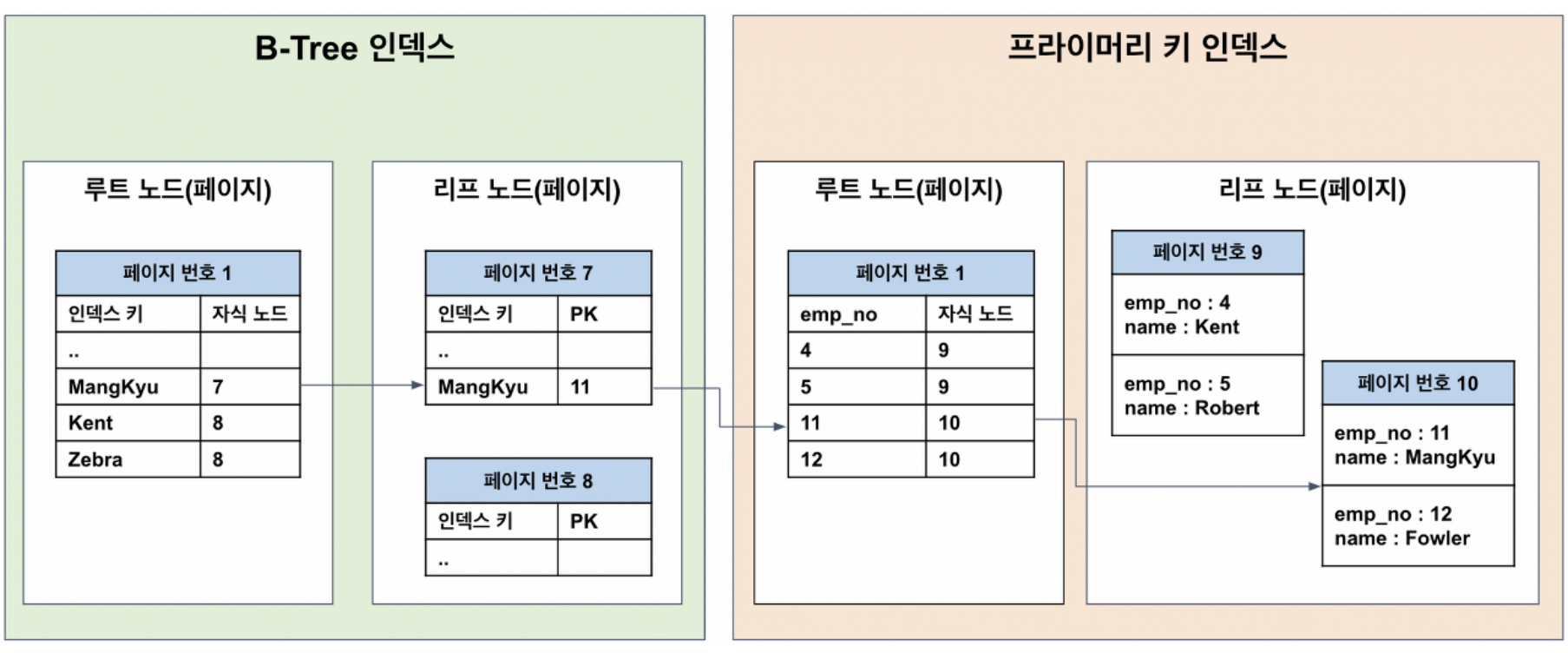
B-Tree

- 이진 트리(Binary Tree)에서 확장된 형태, **N개의 자식**을 가질 수 있음
- **좌우 레벨이 항상 균형을 유지**하는 정렬된 트리



- 가장 일반적인 인덱스, 특수한 경우가 아니면 **대부분 B-Tree 인덱스 사용**
- 인덱스 키를 바탕으로 **항상 정렬된 상태**를 유지
- 정렬된 인덱스 키를 따라서 리프 노드에 도달하면 **(인덱스 키, PK) 쌍으로 저장**되어 있음

```
CREATE TABLE employee (
    emp_no INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(64),
    PRIMARY KEY(emp_no),
    INDEX idx_name (name)
) ENGINE=InnoDB;
```



- 왼쪽 : **B-Tree 인덱스 영역** / 오른쪽 : **프라이머리 키 인덱스(클러스터 인덱스) 영역** 또는 **테이블 영역**

- **모든 페이지는 키 값을 기준으로 정렬**되어 있다. (idx_name 인덱스의 경우, name 값을 기준으로 정렬되어 있다.)
- 데이터를 따라 리프노드에 도달하면 인덱스 키에 해당하는 레코드의 PK 값이 저장되어 있다.
- 인덱스는 테이블과 독립적인 저장 공간이므로 인덱스를 통해 데이터를 조회하려면 먼저 PK를 찾아야 한다.
- PK로 레코드를 조회할 때는(인덱스 영역에서 테이블 영역으로 넘어가는 경우) PK가 어느 페이지에 저장되어 있는지 알 수 없으므로 랜덤 I/O가 발생
- 이후 PK를 따라 리프노드에서 실제 레코드를 읽어온다.
- 참고로 연속된 데이터를 조회하는 경우에는 순차 I/O가 발생하는데, 랜덤 I/O는 임의의 장소에서 데이터를 가져오지만 순차 I/O는 다음 장소에서 데이터를 가져오므로 훨씬 빠르다.

B+ Tree

- B-Tree를 개선시킨 자료구조
- 리프노드(데이터노드)만 인덱스와 함께 데이터(Value)를 가지고 있고, 나머지 노드(인덱스노드)들은 데이터를 위한 인덱스(Key)만을 갖는다.
- 리프노드들은 LinkedList로 연결되어 있다.
- 데이터 노드 크기는 인덱스 노드의 크기와 같지 않아도 된다.

▼ 참고(클러스터 인덱스, 페이지)

클러스터 인덱스

- 클러스터링 : 유사한 것들을 묶는 것
- MySQL은 PK를 기준으로 유사한 값들이 함께 조회되는 경우가 많다는 점에서 착안하여, PK가 유사한 레코드들끼리 묶어서 저장
- PK는 클러스터 인덱스(Clustered Index)라고도 불리며, 그 외의 일반적인 인덱스는 논클러스터 인덱스로 불린다.
- 클러스터링 특성 때문에 레코드의 저장이나 PK의 변경은 처리 속도가 느리다.
  - 레코드를 추가하기 위해 PK 기반으로 레코드의 저장 위치를 탐색해야 하기 때문
  - PK를 변경하는 것은 레코드가 저장된 물리적인 위치를 변경하는 작업이 수반되기 때문
  - 하지만 그럼에도 불구하고 이러한 특성을 갖는 이유는 쓰기 작업을 희생해서라도 빠르게 읽기 작업을 처리하기 위함이다. 읽기 작업이 더욱 우선시 되는 이유는 일반적인 온라인 환경에서 읽기와 쓰기의 비율이 8:2, 9:1 정도이기 때문이다.
- 장점
  - PK로 검색할 때 처리가 매우 빠름
  - 연속되는 PK로 조회할 경우 랜덤 I/O가 아닌 순차 I/O를 사용하여 처리 속도가 더욱 빠름
  - 인덱스가 PK값을 가지므로 인덱스로 PK 값만 조회하는 경우 효율적으로 처리될 수 있음(=커버링 인덱스)
- 단점
  - 모든 인덱스가 PK에 의존하므로 PK 값이 클 경우 전체적으로 인덱스의 크기가 커지고, 페이지 양이 많아짐
  - 인덱스를 통해 검색할 때 PK로 다시 한번 검색해야 하므로 처리 성능이 느림
  - INSERT 시에 PK에 의해 레코드의 저장 위치가 결정되기 때문에 처리 성능이 느림
  - PK를 변경할 때 레코드를 DELETE 및 INSERT 해야 하므로 처리 성능이 느림

페이지

- 디스크와 메모리(버퍼풀)에 데이터를 읽고 쓰는 최소 작업 단위
- 만약 쿼리를 통해 1개의 레코드를 읽고 싶더라도 하나의 블록을 읽어야 함
- 페이지에 저장되는 개별 데이터의 크기를 **최대한 작게** 하여, **1개의 페이지에 많은 데이터들을 저장**할 수 있도록 하는 것이 상당히 중요
- **페이지에 저장되는 데이터의 크기가 크면**
  - 디스크 I/O가 많아질 수 있음

- 메모리에 캐싱할 수 있는 페이지의 수가 줄어들 수 있음