

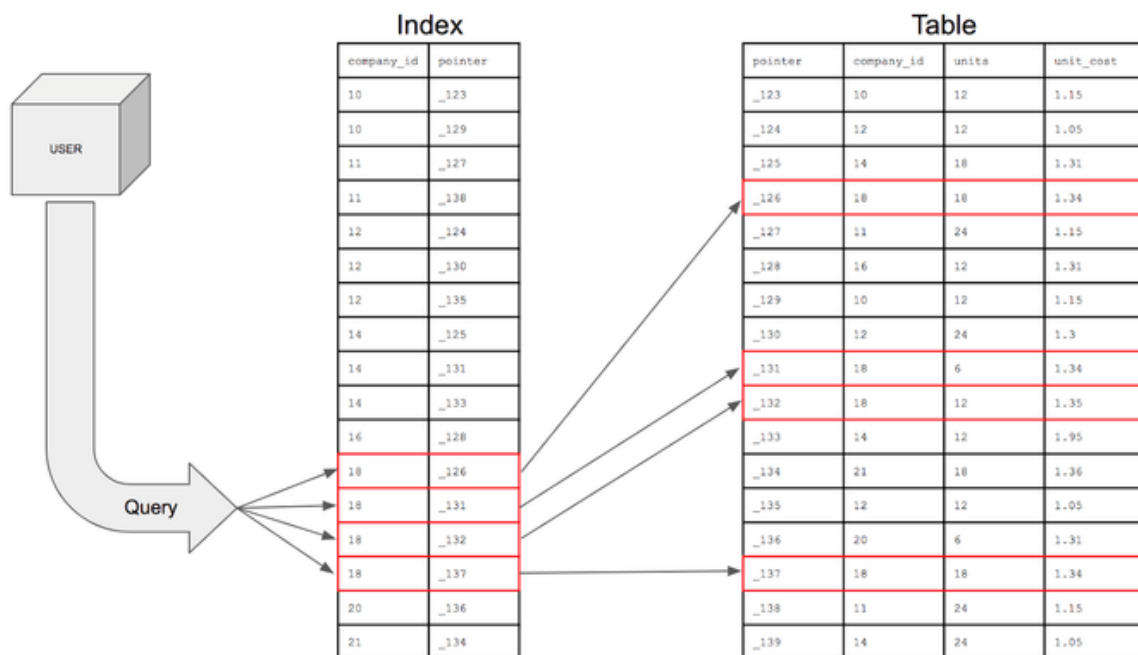
[DB] Index

(1) 인덱스(Index)란?



인덱스(index)란 자료를 쉽고 빠르게 찾을 수 있도록 만든 데이터 구조로 데이터베이스에서 인덱스란 원하는 **RDBMS 에서 검색 속도를 높이기 위한 기술**로 추가적인 쓰기 작업과 저장 공간을 활용하여 테이블 칼럼을 색인화한 자료구조이다.

쉽게 말하자면, 책에 있는 목차를 생각하면 된다. 책에 있는 내용을 찾기 위해서, 먼저 목차에서 찾고 목차에 있는 페이지 번호를 보고 찾아가 원하는 내용을 찾듯이 인덱스에서 내가 원하는 데이터를 먼저 찾고 저장되어 있는 물리적 주소로 찾아가는 것이다.



예를 들어, 인덱스 기준이 하나도 잡혀 있지 않아 이름, 성별, 이메일 등이 담긴 회원 테이블이 입력 된 순서대로 존재한다고 가정해보자. 이때 이메일로 회원을 찾으려면 전체 데이터에서 순차적으로 찾는 이메일이 나올 때까지 검색을 하게 된다.

여기서 인덱스를 이메일로 정했다면, 이메일을 기준으로 정렬이 되게 됩니다. 또 다시 이메일로 회원을 찾으려고 한다면, 아까보다 속도가 훨씬 빨라질 것입니다.

즉, 인덱스의 특징으로는

- (1) 인덱스는 항상 최신의 정렬 상태를 유지
- (2) 인덱스도 하나의 데이터베이스 객체
- (3) 데이터베이스 크기의 약 10% 정도의 저장 공간 필요가 있다.

(2) 인덱스 알고리즘

1. Full Table Scan (인덱스 적용 X)

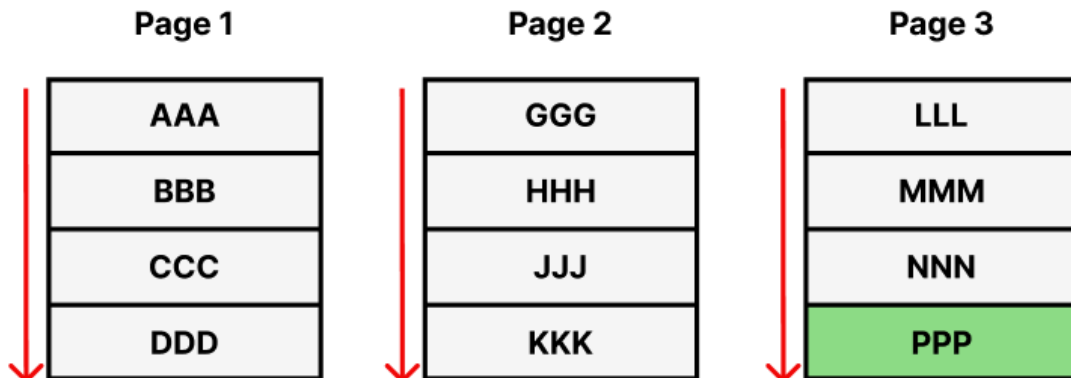


Full Table Scan(순차 접근)이란 테이블에 존재하는 모든 데이터를 읽어 가면서 조건에 맞으면 결과로서 추출하고 조건에 맞지 않으면 버리는 방식이다.

Full Table Scan 을 사용하는 경우는 (1) 적용 가능한 인덱스가 없는 경우 (2) 인덱스 처리 범위가 넓은 경우(적용 가능한 인덱스가 존재하더라도 처리 범위가 넓어서 Full Table Scan이 더 적은 비용이 든다면 ..) (3) 크기가 작은 테이블에 액세스 하는 경우 등이 있다.

Full Table Scan

'PPP'를 찾고 싶어! ⇒ 총 3개의 페이지, 12번의 검색을 통해 알 수 있음

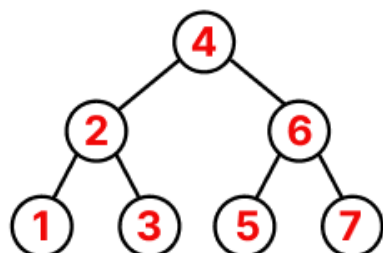


2. B-Tree(Balanced Tree)

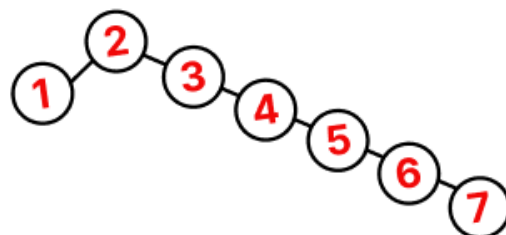
Binary Search Tree

Binary Search Tree (이진 탐색 트리)의 단점을 극복하기 위해 나온 것 중 하나가 바로 B-Tree다.

먼저, Binary Search Tree를 살펴보면, 이진 트리의 경우 각각의 노드가 최대 두 개의 자식 노드를 가지는 트리 자료구조이다. 균형 있는 이진 탐색 트리의 경우 검색 시간 복잡도가 $O(\log N)$ 이지만, 균형 없는 이진 탐색 트리의 경우 시간 복잡도는 $O(n)$ 이다. 즉, '운이 나쁘면..' 전체를 다 탐색해야 할 수도 있다는 소리다.



균형 있는 이진탐색트리

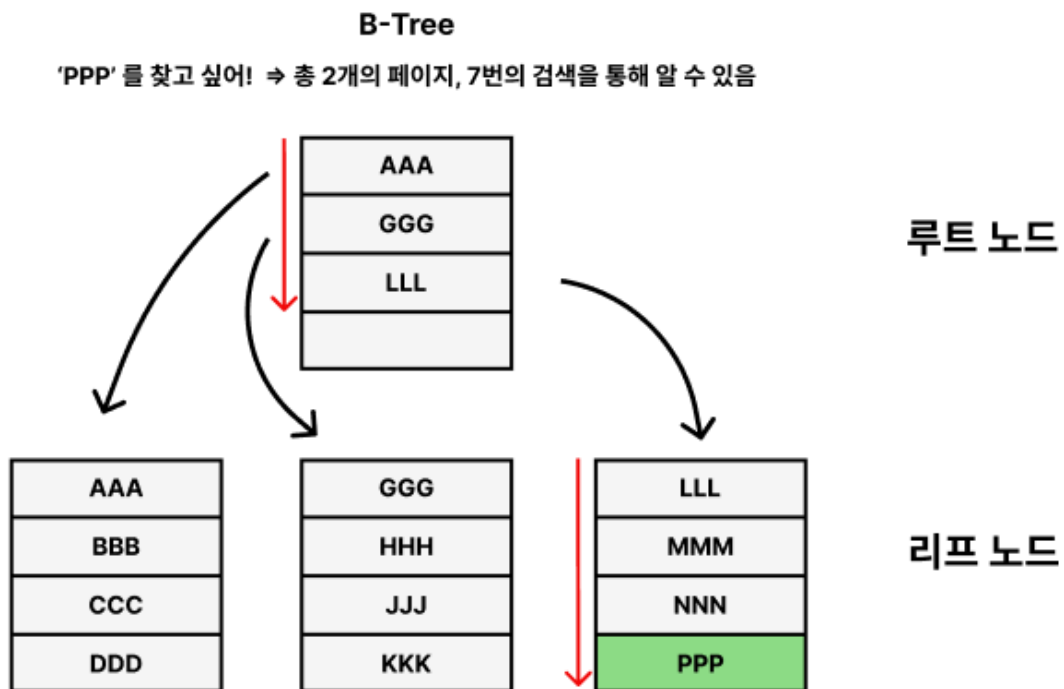


균형 없는 이진탐색트리

이러한 단점을 극복하기 위해 나온 것이 B-Tree 다.

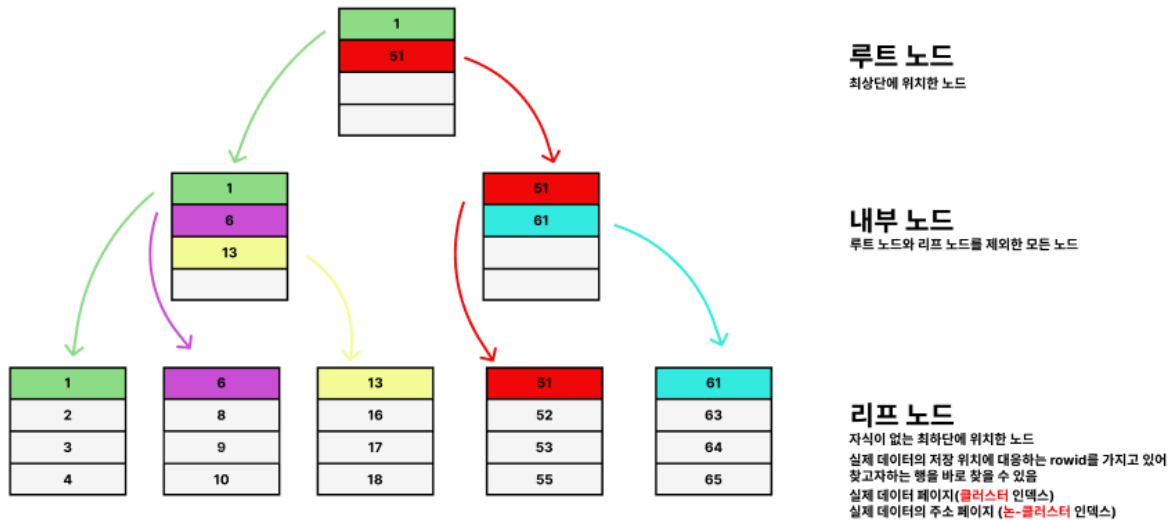
B-Tree

아까 Full Table Scan으로 진행했던 것을 B-Tree로 진행을 한다면 아래와 같게 된다.



B-Tree: 대부분의 RDBMS 는 B-Tree 구조로 되어 있다. 데이터의 검색 시간을 단축하기 위한 자료구조로, 루트 노드, 내부 노드, 리프 노드로 구성되며, 리프 노드가 모두 같은 레벨에 존재하는 균형(Balanced) 트리로 $O(\log N)$ 시간 복잡도를 가진다.

B-Tree(Balanced Tree)



B-Tree 특징

- (1) 모든 노드는 최대 m개의 자식들을 가진다. (2개 이상의 자식 노드를 가질 수 있다.)
- (2) 키 값은 오름차순으로 저장
- (3) 모든 리프노드들은 같은 높이에 있어야 한다.

B-Tree 의 장점

- 균형된 트리 구조를 이용하기 때문에 한 번 검색할 때마다 검색 대상이 줄어들어 접근 시간이 적게 걸린다.
- 노드의 삽입/삭제 후에도 균형 트리 유지로 균등한 응답 속도를 보장한다.

B-Tree 의 단점

- B-Tree 의 경우에는, 노드의 높이가 한정되어 있기 때문에 데이터를 삽입하려고 할 때 해당 노드가 꽉 차 있다면 동적으로 노드를 분할하게 된다. (비어있는 노드를 확보한 다음 문제가 되는 노드에 데이터를 공평하게 나누어 저장)
- 또한, 데이터를 삭제하려고 할 때도 리프 노드의 경우에는 삭제하고, 최소 키를 가지지 않았다면 다시 재조정하게 된다.

→ 즉, 삽입/삭제를 할 때 처리할 대상을 찾기 위한 조회 성능은 향상 하겠지만 B-tree의 모양을 유지하기 위해 노드의 분할 및 이동이 자주 발생해 성능이 떨어지게 된다.



즉, SELECT 문에 대한 성능은 향상되지만 INSERT UPDATE DELETE 문에서는 성능이 저하된다. 이러한 단점에도 불구하고 일반적인 RDBMS 에서 B-Tree를 쓰는 이유는 일반적인 웹서비스에서 쓰기과 읽기의 비율이 2:8 ~ 1:9 정도 되기 때문이다.

(3) 인덱스 종류



인덱스의 종류에는 클러스터 인덱스와 논-클러스터 인덱스가 존재하고, 모두 B-Tree 인덱스를 기본으로 한다.

예를 들어, id 를 pk 로 가지고, name, email 을 가지는 테이블을 만들어보자. 여기서 email 은 중복이 없게 할 것이다.

```
CREATE TABLE member (  
  id      int      primary key,    --> 클러스터 인덱스 자동 생성  
  name    varchar(255),  
  email   varchar(255) unique      --> 논-클러스터 인덱스 자동 생성  
);
```

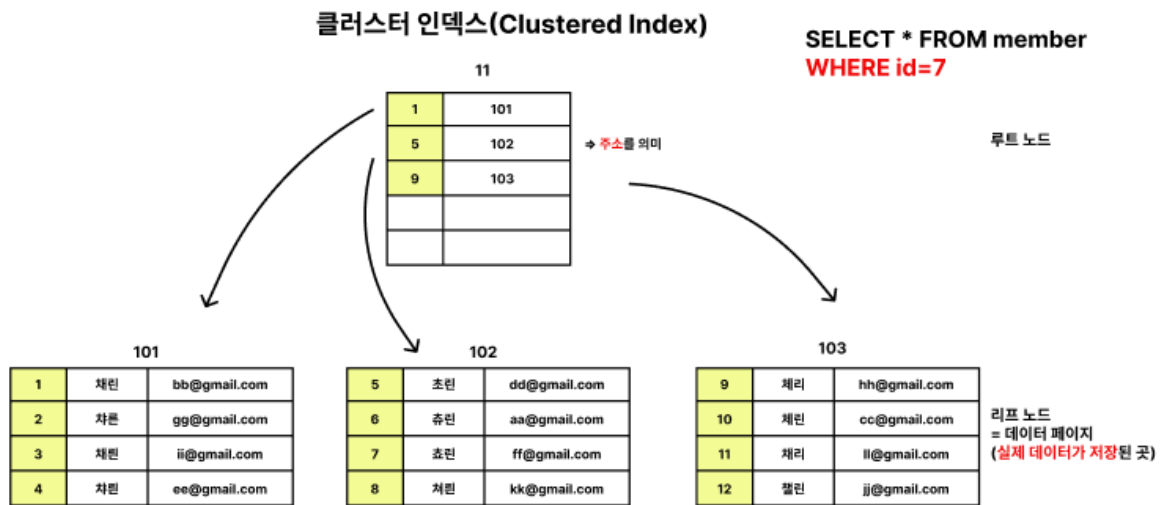
이렇게 하게 되면 pk 로 설정한 id 에 클러스터 인덱스가 자동 생성되고 unique로 설정한 email 에 논-클러스터가 자동 생성된다.

1. 클러스터 인덱스



클러스터 인덱스(Clustered Index) 는 테이블 당 하나만 생성할 수 있으며, 연속된 키 값의 레코드를 묶어서 같은 블록에 저장하는 방법으로 **B-tree 인덱스의 리프 노드에서 페이지의 주소 값 대신 테이블의 열 자체가 저장되는 형태이다.**

예를 들면, id 를 pk 로 갖는 id, name, email이 담긴 테이블이 있다고 가정을 해보면 id 를 클러스터 인덱스로 가지는 B-tree 가 생성이 된다.



클러스터 인덱스의 특징

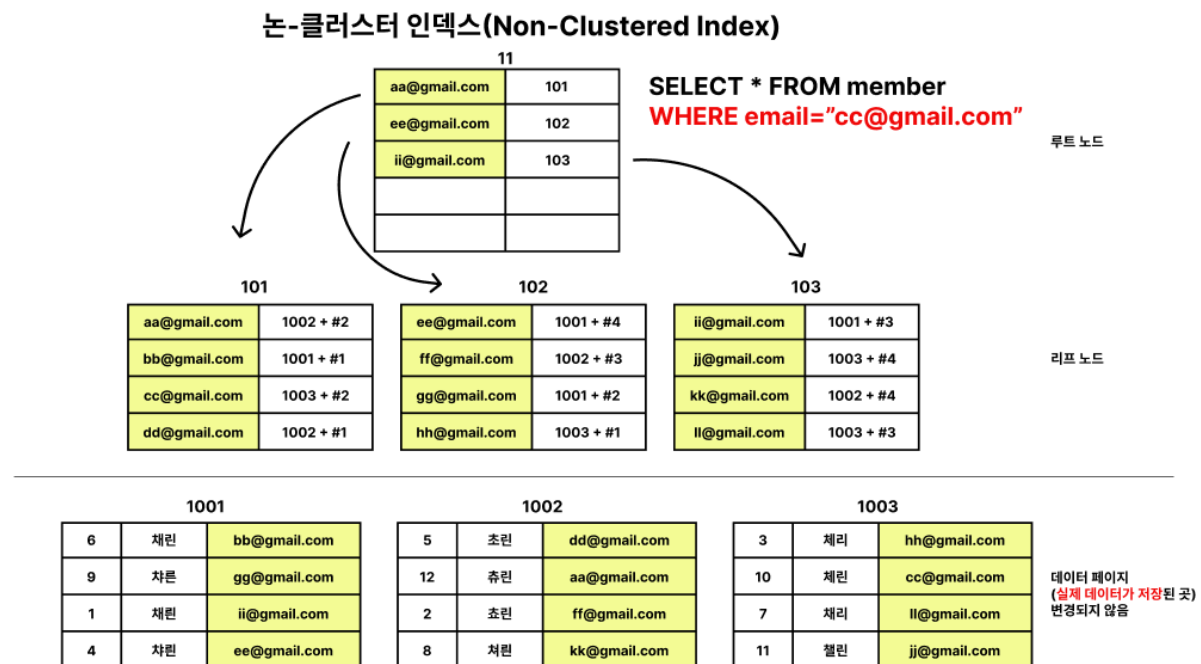
- 테이블당 1개씩만 허용한다.
- pk 설정 시 그 칼럼은 자동으로 클러스터 인덱스가 만들어진다. (unique + not null 도 만들어지지만, pk 보다 우선순위가 낮음)
- 인덱스 자체의 리프 페이지가 곧 데이터다. (리프 페이지에는 따로 인덱스 페이지를 만들지 않음)
- 데이터 입력, 수정, 삭제 시 항상 정렬 상태를 유지한다.
- 논-클러스터 인덱스보다 검색 속도는 더 빠르지만 데이터의 입력, 수정, 삭제는 느리다.
- 키 값에 의한 동등 및 범위(BETWEEN) 검색 모두에 유리하다.
- 인덱스 저장 시 차지하는 공간이 적다.

2. 논-클러스터 인덱스 (= Secondary Index, 보조 인덱스)



논-클러스터 인덱스(Non-Clustered Index)란 속성의 값으로 B-tree 인덱스를 구성하며 리프 노드의 각 행은 해당 페이지의 주소 값을 저장한다.

email 속성으로 인덱스를 생성한 경우에 대해 알아보자 (여기서 id 는 pk 값이 아니라고 가정)



논-클러스터 인덱스의 특징

- 테이블당 여러개를 만들 수 있음 (복합 인덱스 가능)
- 레코드의 원본은 정렬되지 않고, 인덱스 페이지만 정렬된다.
- 인덱스 자체의 리프 페이지는 데이터가 아니라 데이터가 위치하는 포인터(RID)이기 때문에 클러스터형보다 검색 속도는 더 느리지만 데이터의 입력, 수정, 삭제는 더 빠르다.
- 특정 키 값을 찾는 검색의 경우는 성능을 보장할 수 있지만, 범위 검색 (BETWEEN) 검색의 경우는 저장 순서가 일정치 않을 수 있어 빠른 검색을 보장하지는 못한다.
- 인덱스를 생성할 때 데이터 페이지는 그냥 둔 상태에서 별도의 인덱스 페이지를 따로 만들기 때문에 용량을 더 차지한다.

cf) RID(Row ID) 란 데이터 위치 포인터를 뜻하는 것으로 해당하는 행의 '데이터 페이지 번호-행의 순번(데이터 페이지 오프셋)' 으로 구성되는 포인팅 정보임

클러스터 인덱스 + 논-클러스터 인덱스

하지만, 우리는 클러스터 인덱스와 논-클러스터 인덱스를 보통 같이 사용한다. 예를 들어 id 는 클러스터 인덱스로, email 은 논-클러스터 인덱스로 사용해서 id 와 email 로 검색하는 경우를 생각해보자

aa@gmail.com	101
ee@gmail.com	102
ii@gmail.com	103

SELECT * FROM member
WHERE email="cc@gmail.com"

101

aa@gmail.com	6
bb@gmail.com	1
cc@gmail.com	10
dd@gmail.com	5

102

ee@gmail.com	4
ff@gmail.com	7
gg@gmail.com	2
hh@gmail.com	9

103

ii@gmail.com	3
jj@gmail.com	12
kk@gmail.com	8
ll@gmail.com	11

email
논-클러스터 인덱스 페이지

id
클러스터 인덱스 페이지

1	1001
5	1002
9	1003

1001

1	채린	bb@gmail.com
2	차론	gg@gmail.com
3	채린	ii@gmail.com
4	차린	ee@gmail.com

1002

5	초린	dd@gmail.com
6	츄린	aa@gmail.com
7	초린	ff@gmail.com
8	쳐린	kk@gmail.com

1003

9	체리	hh@gmail.com
10	체린	cc@gmail.com
11	채리	ll@gmail.com
12	첼린	jj@gmail.com

(4) 인덱스 적용 기준

- 그렇다면 인덱스를 어떤 컬럼에 적용해야 할까?
 - Cardinality (그룹 내 요소의 개수) 가 높은 것 = 즉, 중복 수치가 낮은 것
 - WHERE, JOIN, ORDER BY 절에 자주 사용되는 컬럼 (조건 절이 없다면 인덱스가 사용되지 않음)
 - INSERT, UPDATE, DELETE 가 자주 발생하지 않는 컬럼 (성능 저하에 대한 영향이 있을 수 있기 때문)
 - 규모가 작지 않은 테이블

- cf. **Cardinality**

- 예를 들어, id / 이름 / 성별 / 이메일 / 사는 지역 / 나이 등이 있는 member table 이 있다고 하자. 여기서 요소의 개수가 가장 많은 것은, id & 이메일(unique 라면)이 될 것이다.
 - 성별은 남 / 여 2가지 밖에 없을 것

참고

<https://junghn.tistory.com/entry/DB-클러스터-인덱스와-넌클러스터-인덱스-개념-총정리>

MySQL 로 배우는 데이터베이스 개론과 실습