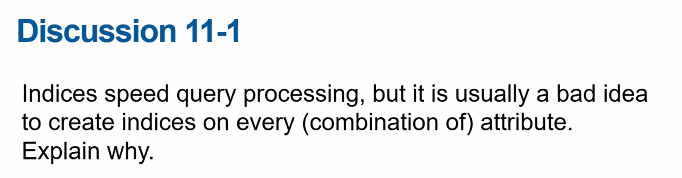
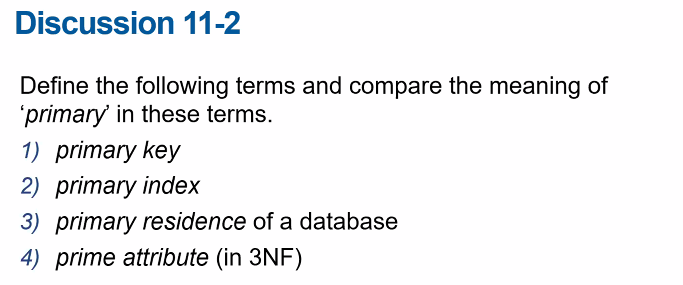
**Discussion 05/20**



Index를 만드는 것은 약간의 space overhead, insertion/deletion overhead를 감수하고 access를 빠르게 하기 위함이다. 하지만 모든 attribute에 index를 만들 경우 primary key를 제외하면 sequential로 저장되어 있지 않기 때문에 dense index가 되어야 한다. 따라서 space나 insertion/deletion의 overhead가 access의 효율성이 높아지는 것보다 더 크다.

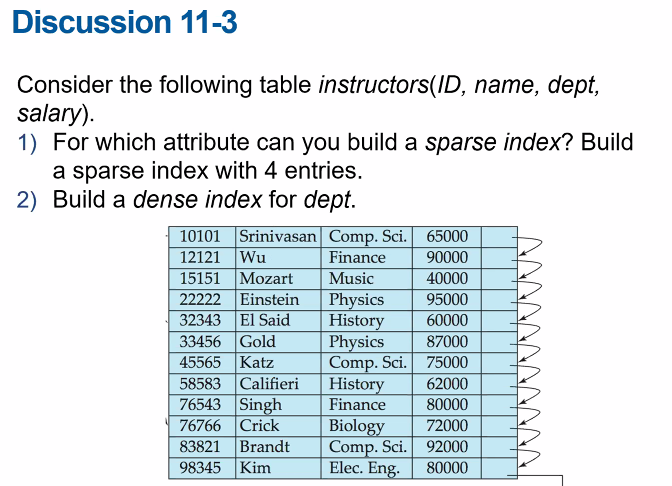


1. relation에서 유일성, 최소성을 보장하는 candidate key 중 중요하다고 생각해 고른 키.

2. index 중에서 search key가 파일에서 순서대로 저장되어 있는 인덱스.

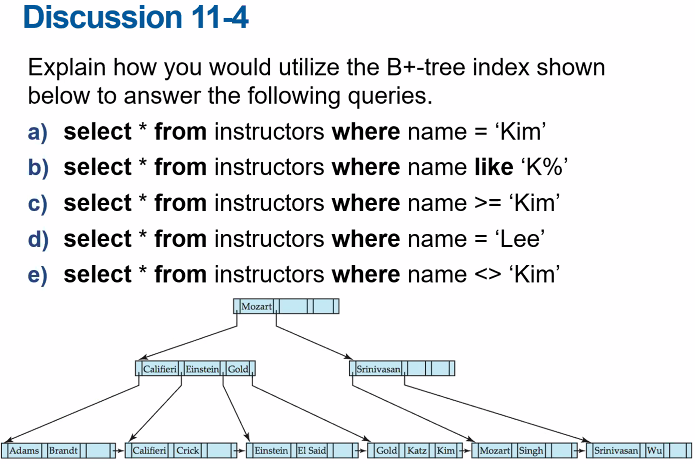
3. 초창기에 disk를 표현하던 용어.

4. a->b에서 (b-a)가 candidate key에 포함되는 경우 이를 prime attribute라고 함. candidate key의 attribute들 중 유일성을 결정하는 attribute.



1. id. Id만 정렬되어 있기 때문. Id에 대해 4개의 entry로 만들면 10101 / 22222 / 45565 / 76766

2. Comp. Sci. / Finance / Music / Physics / History / Biology / Elec. Eng. 가 각각 bucket을 가리키고 bucket에서 각각의 record를 가리키게 함.



a) root에서부터 비교할 때 Kim이 각각 node보다 크거나 작은지를 파악해서 해당 위치까지 탐색. Kim이 나오면 반환하고 Kim보다 큰 값이 나오면 없으니까 없다고 함.

b) K가 각각 노드보다 크거나 작은지를 파악하고, leaf node에서 순차적으로 앞자리가 K가 아닐 때까지 오른쪽으로 모두 탐색

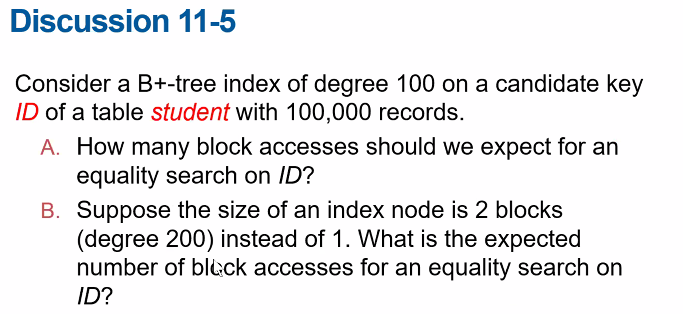
c) 각각 node에서 Kim 과 대소비교 한 다음, Kim을 가진 leaf node에서부터 오른쪽 끝까지 순차적으로 탐색.

d) a)와 마찬가지 방법을 Lee에 대해서 하면 됨.

e) a) 방법으로 Kim을 찾은 다음 그 노드를 제외하고 그 왼쪽, 오른쪽 방향으로 끝까지 모두 탐색.

=> 근데 e)는 이 방법이 비효율적이라서 굳이 index를 이용하지 않아도 됨.

=> )b에서 %K% 이런 검색은 index를 사용할 수가 없음.



A. 트리 높이가 log\_(100/2)(100000) 이기 때문에 3이다. 따라서 평균 3번의 block access가 예상된다.

=> record block access까지 계산해야 함. 따라서 +1을 해서 4번.

B. 트리 높이가 log\_(200/2)(100000) 인데 마찬가지로 3이다. 근데 index node가 2개의 block으로 구성되어 있으므로 leaf node까지 6번의 block access가 필요하다.

=> record block access까지 계산하면 6+1 해서 7번.