Discussion 05/20

Discussion 11-1

Indices speed query processing, but it is usually a bad idea to create indices on every (combination of) attribute. Explain why.

Index 를 만드는 것은 약간의 space overhead, insertion/deletion overhead 를 감수하고 access 를 빠르게 하기 위함이다. 하지만 모든 attribute 에 index 를 만들 경우 primary key 를 제외하면 sequential 로 저장되어 있지 않기 때문에 dense index 가 되어야 한다. 따라서 space 나 insertion/deletion 의 overhead 가 access 의 효율성이 높아지는 것보다 더 크다.

Discussion 11-2

Define the following terms and compare the meaning of '*primary*' in these terms.

1) *primary key*
2) *primary index*
3) *primary residence* of a database
4) *prime attribute* (in 3NF)

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

2. index 중에서 search key 가 파일에서 순서대로 저장되어 있는 인덱스.

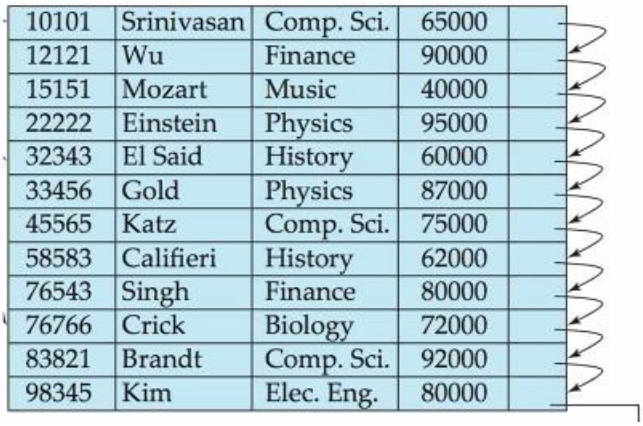
3. 초창기에 disk 를 표현하던 용어.

4. a->b 에서 (b-a)가 candidate key 에 포함되는 경우 이를 prime attribute 라고 함. candidate key 의 attribute 들 중 유일성을 결정하는 attribute.

Discussion 11-3

Consider the following table *instructors*(*ID, name, dept, salary*).

1) For which attribute can you build a *sparse index*? Build a sparse index with 4 entries.
2) Build a *dense index* for *dept*.



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

1. id. Id 만 정렬되어 있기 때문. Id 에 대해 4 개의 entry 로 만들면 10101 / 22222 / 45565 / 76766

2. Comp. Sci. / Finance / Music / Physics / History / Biology / Elec. Eng. 가 각각 bucket 을 가리키고 bucket 에서 각각의 record 를 가리키게 함.

Discussion 11-4

Explain how you would utilize the B+-tree index shown below to answer the following queries.

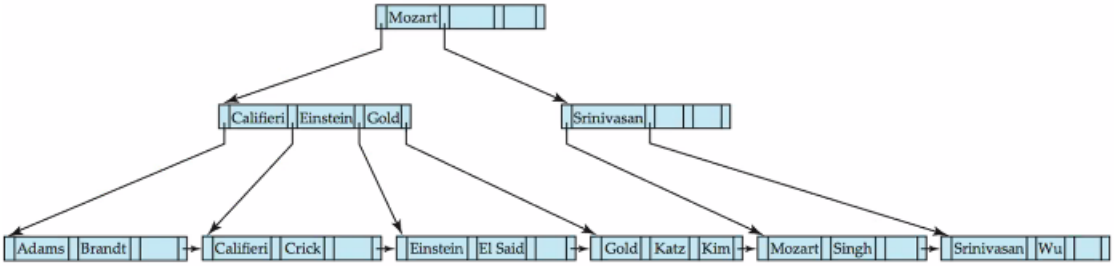
a) **select** * **from** instructors **where** name = 'Kim'

b) **select** * **from** instructors **where** name **like** 'K%'

c) **select** * **from** instructors **where** name >= 'Kim'

d) **select** * **from** instructors **where** name = 'Lee'

e) **select** * **from** instructors **where** name <> 'Kim'



a) root 에서부터 비교할 때 Kim 이 각각 node 보다 크거나 작은지를 파악해서 해당 위치까지 탐색. Kim 이 나오면 반환하고 Kim 보다 큰 값이 나오면 없으니까 없다고 함.

b) K 가 각각 노드보다 크거나 작은지를 파악하고, leaf node 에서 순차적으로 앞자리가 K 가 아닐 때까지 오른쪽으로 모두 탐색

c) 각각 node 에서 Kim 과 대소비교 한 다음, Kim 을 가진 leaf node 에서부터 오른쪽 끝까지 순차적으로 탐색.

d) a)와 마찬가지 방법을 Lee 에 대해서 하면 됨.

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

=〉 근데 e)는 이 방법이 비효율적이라서 굳이 index 를 이용하지 않아도 됨.

=〉)b 에서 %K% 이런 검색은 index 를 사용할 수가 없음.

Discussion 11-5

Consider a B+-tree index of degree 100 on a candidate key *ID* of a table *student* with 100,000 records.

A. How many block accesses should we expect for an equality search on *ID*?

B. Suppose the size of an index node is 2 blocks (degree 200) instead of 1. What is the expected number of block accesses for an equality search on *ID*?

A. 트리 높이가 log_(100/2) (100000) 이기 때문에 3 이다. 따라서 평균 3 번의 block access 가 예상된다.

=〉 record block access 까지 계산해야 함. 따라서 +1 을 해서 4 번.

B. 트리 높이가 log_(200/2) (100000) 인데 마찬가지로 3 이다. 근데 index node 가 2 개의 block 으로 구성되어 있으므로 leaf node 까지 6 번의 block access 가 필요하다.

=〉 record block access 까지 계산하면 6+1 해서 7 번.