14장: 인덱싱

Chapter 14: Indexing

데이터베이스 시스템 개념, 7 판.

개요

Outline

- § 기본 개념
- § 정렬된 인덱스
- § B+-트리 인덱스 파일
- § B-Tree 인덱스 파일
- § 해싱 § 쓰
- 기 최적화 인덱스 § 시공간 인덱싱

기본 개념 Basic Concepts

§ 원하는 데이터에 대한 액세스 속도를 높이는 데 사용되는 인덱싱 메커니즘.

- 예: 라이브러리의 저자 카탈로그

§ 검색 키 – 파일에서 레코드를 조회하는 데 사용되는 속성 집합입니다. § 색인 파일은 다음 형식의 레코드(색인 항목이라고 함) 로 구성됩니다.

검색 키	포인터

§ 색인 파일은 일반적으로 원본 파일보다 훨씬 작습니다.

§ 인덱스의 두 가지 기본 종류:

- 정렬된 인덱스: 검색 키가 정렬된 순서로 저장됩니다.
- 해시 인덱스: 검색 키가 전체에 균일하게 배포됩니다.
  "해시 함수"를 사용하는 "버킷".

지수 평가 지표 Index Evaluation Metrics

§ 효율적으로 지원되는 액세스 유형. 예를 들어

- 속성에 지정된 값이 있는 레코드
- 지정된 값 범위에 속하는 속성 값이 있는 레코드.

§ 액세스 시간

§ 삽입 시간

§ 삭제 시간

§ 공간 오버헤드

정렬된 인덱스 Ordered Indices

! 정렬된 인덱스 에서 인덱스 항목은 검색 키 값에 따라 정렬되어 저장됩니다.

! 클러스터링 인덱스: 순차적으로 정렬된 파일에서 검색 키가 파일의 순차적 순서를 지정하는 인덱스입니다.

! 기본 인덱스 라고도 합니다 ! 클러스터링 인덱스의 검색 키는 일반적으로 기본 키이지만 반드시 그런 것은 아닙니다.

! 보조 인덱스: 검색 키가 파일의 순차적 순서와 다른 순서를 지정하는 인덱스입니다. 비클러스터링 인덱스 라고도 합니다 .

! 인덱스 순차 파일: 순차 검색 키에 클러스터링 인덱스가 있는 검색 키에 정렬된 파일입니다.

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	

Figure 14.1 Sequential file for *instructor* records.

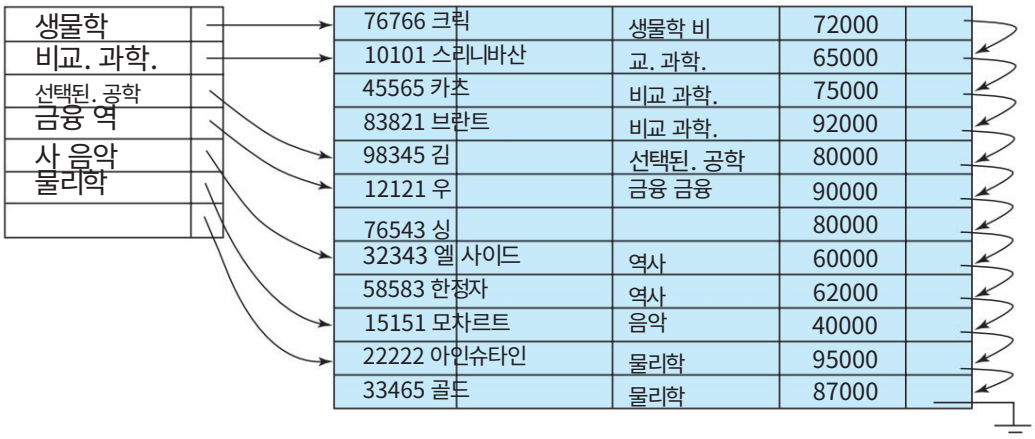
조밀한 인덱스 파일 Dense Index Files

§ 조밀한 인덱스 — 인덱스 레코드는 모든 검색 키 값에 대해 나타납니다. 파일.

§ 예: 강사 관계의 ID 속성에 대한 인덱스

조밀한 인덱스 파일(계속)(Cont.)

§ dept_name에 대한 조밀한 인덱스, dept_name에 정렬된 강사 파일 포함



클러스터링 및 비클러스터링 인덱스

§ 인덱스는 레코드를 검색할 때 상당한 이점을 제공합니다. § 그러나: 인덱스는 데이터베이스 수정에 오버헤드를 부과합니다.

- 레코드가 삽입되거나 삭제될 때 관계의 모든 인덱스는 업데이트
- 레코드가 업데이트되면 업데이트된 속성에 대한 모든 인덱스는 업데이트

§ 클러스터링 인덱스를 사용하는 순차 스캔은 효율적이지만 보조(비클러스터링) 인덱스를 사용하는 순차 스캔은 자기 디스크에 비용이 많이 듭니다.

- 각 레코드 액세스는 디스크에서 새 블록을 가져올 수 있습니다. • 자기 디스크의 각 블록 가져오기에는 약 5~10밀리초가 필요합니다.

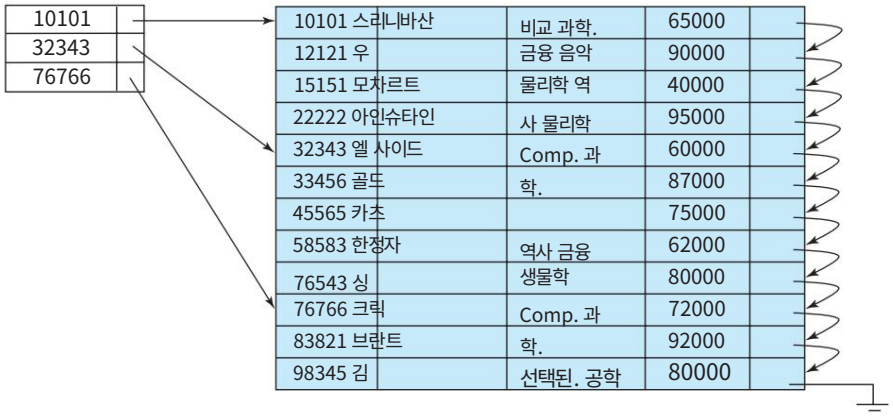
스파스 인덱스 파일

§ 희소 인덱스: 일부 검색 키에 대한 인덱스 레코드만 포함
가치.

- 검색키에 레코드가 순차적으로 정렬된 경우에 해당한다.

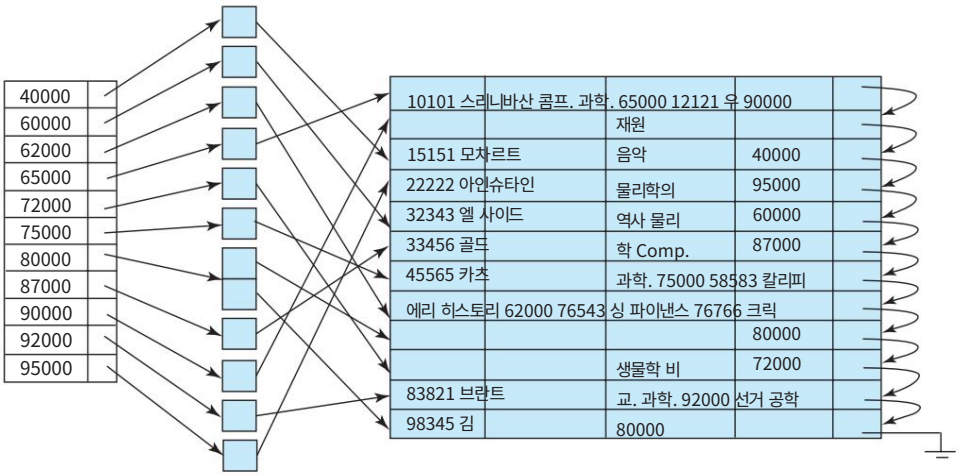
§ 검색 키 값 K가 있는 레코드를 찾으려면 다음을 수행합니다.

- 가장 큰 검색 키 값 <= K 인 인덱스 레코드 찾기 • 인덱스 레코드가 가리키는 레
코드부터 순차적으로 파일 검색



보조 인덱스 예시

§ 강사의 급여 분야에 대한 보조 지표



§ 인덱스 레코드는 특정 검색 키 값을 가진 모든 실제 레코드에 대한 포인터를 포함하는 버킷을 가리킵
니다.

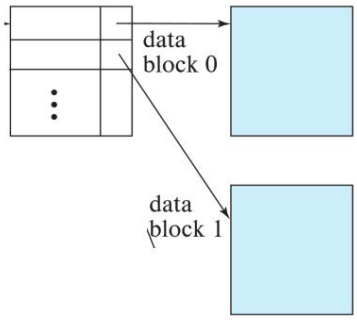
§ 보조 인덱스는 밀도가 높아야 합니다.

스파스 인덱스 파일(계속)

§ 고밀도 인덱스와 비교: • 공간이 적고 삽입 및 삭제를 위한 유지 관리 오버헤드가 적습니다. • 레코드를 찾기 위해 일반적으로 밀집 인덱스보다 느립니다.

§ 좋은 트레이드오프:

• 클러스터형 인덱스의 경우: 파일의 모든 블록에 대한 인덱스 항목이 있는 희소 인덱스, 블록에서 가장 작은 검색 키 값에 해당합니다.



• 비클러스터형 인덱스의 경우: 밀도가 높은 인덱스 위에 있는 희소 인덱스(다단계 인덱스)

다단계 색인

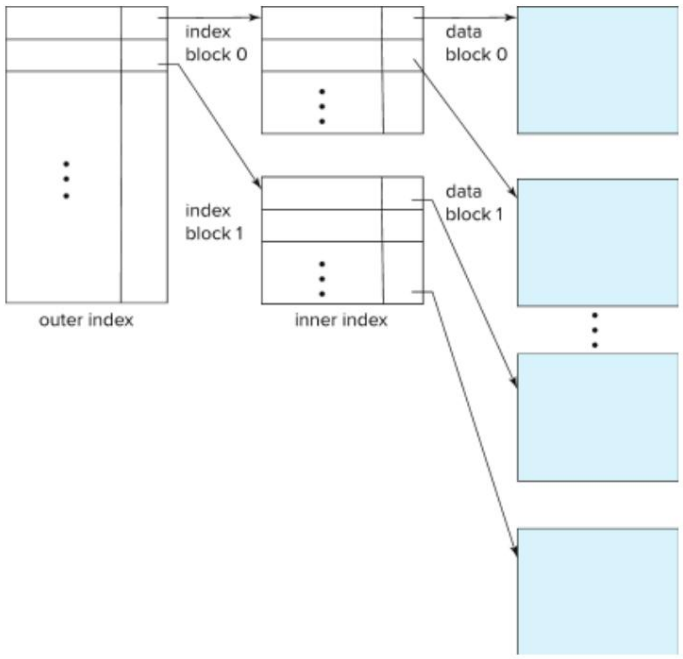
§ 인덱스가 메모리에 맞지 않으면 액세스 비용이 많이 듭니다. § 솔루션: 디스크에 보관된 인덱스를 순차 파일로 취급하고 여기에 스파스 인덱스를 구성합니다.

• 외부 인덱스 – 기본 인덱스의 희소 인덱스
• 내부 인덱스 – 기본 인덱스 파일

§ 외부 색인이 메인 메모리에 맞추기에는 너무 크면 또 다른 수준의 색인을 만들 수 있습니다.

§ 파일에서 삽입하거나 삭제할 때 모든 수준의 색인을 업데이트해야 합니다.

다단계 색인(계속)



인덱스 업데이트: 삭제

§ 삭제된 레코드가 특정 검색 키 값을 가진 파일의 유일한 레코드인 경우 검색 키도 인덱스에서 삭제됩니다.

§ 단일 수준 인덱스 항목 삭제:

- 조밀한 인덱스 – 검색 키 삭제는 파일 레코드와 유사합니다. 삭제.
- 희소 인덱스 –
  - § 검색 키에 대한 항목이 색인에 있으면 색인의 항목을 파일의 다음 검색 키 값으로 대체하여 삭제합니다(검색 키 순서로). § 다음 검색 키 값에 이미 인덱스 항목이 있는 경우 해당 항목은

    교체되지 않고 삭제됩니다.

인덱스 업데이트: 삽입

§ 단일 수준 인덱스 삽입:

- 검색할 레코드의 검색 키 값을 사용하여 조회를 수행합니다.
  삽입.
- 조밀한 인덱스 – 검색 키 값이 인덱스에 나타나지 않으면
  삽입
  - § 인덱스는 순차 파일로 유지됩니다. § 새 항목을 위
    한 공간을 만들어야 하며, 오버플로 블록이
    필수의
- 스파스 인덱스 – 인덱스가 파일의 각 블록에 대한 항목을 저장하는 경우 새 블록이 생성되지 않는 한 인덱스를 변경할 필요가 없습니다. § 새로운 블록이 생성되면 새로운 블록에 처음 나타나는 검색키 값이 인덱스에 삽입된다.

§ 다단계 삽입 및 삭제: 알고리즘은
단일 수준 알고리즘

여러 키의 인덱스

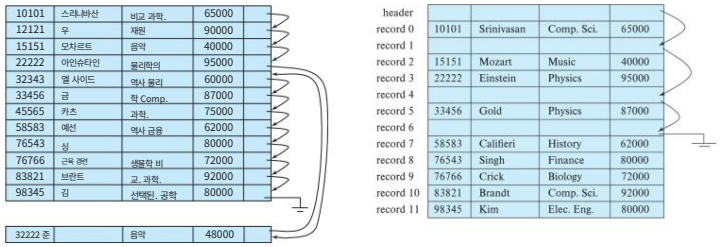
§ 복합 검색 키

- 예, 속성(이름, ID)에 대한 강사 관계 인덱스 • 값은 사전순으로 정렬됩니다.
  - § Eg (John, 12121) < (John, 13514) and (John, 13514) < (Peter, 11223) • 이름만 쿼리
    하거나 (이름, ID) 쿼리 가능

B+-트리 인덱스 파일 B+-Tree Index Files

§ 인덱싱된 순차 파일의 단점 • 파일이 커질수록 성능이 저하됩니다.
만들어진.

• 전체 파일의 주기적인 재구성이 필요합니다.



10101	스리니바산	비교 과학.	65000
12121	우	재원	90000
15151	모차르트	음악	40000
22222	아인슈타인	물리학의	95000
32343	엘 사이드	역사 물리	60000
33456	금	학 Comp.	87000
45565	카츠	과학.	75000
58583	예선	역사 금융	62000
76543	싱		80000
76766	근육 경련	생물학 비	72000
83821	브란트	교. 과학.	92000
98345	김	선택된. 공학	80000
32222 존		음악	48000

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

§ B+-트리 인덱스 파일 의 장점 :

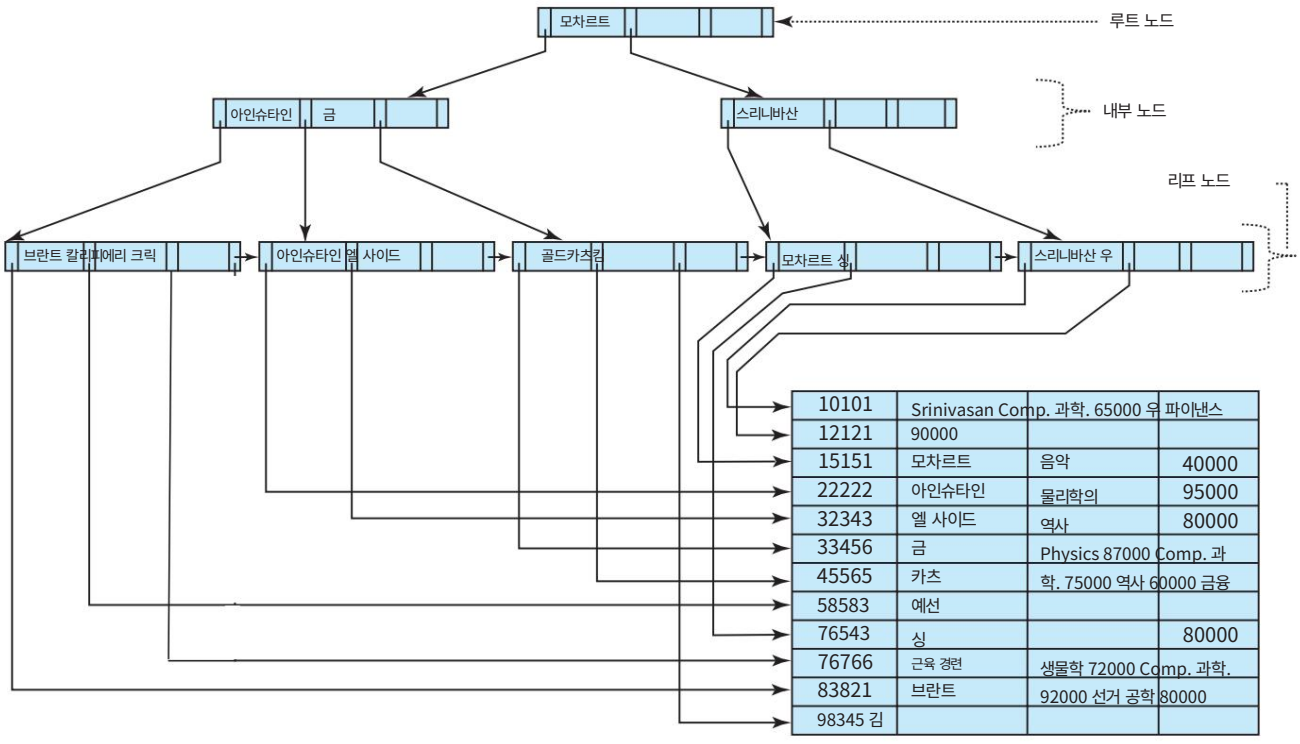
• 삽입 및 삭제에 직면하여 작은 로컬 변경으로 자체적으로 자동 재구성합니다.

• 성능 유지를 위해 전체 파일을 재구성할 필요가 없습니다.

§ B+-트리 의 (사소한) 단점 : • 추가 삽입 및 삭제 오버헤드, 공간 오버헤드.

§ B+-트리 의 장점이 단점보다 큼 • B+-트리가 광범위하게 사용됨

B+-트리 의 예



B+-트리 인덱스 파일(계속) B+-Tree Index Files (Cont.)

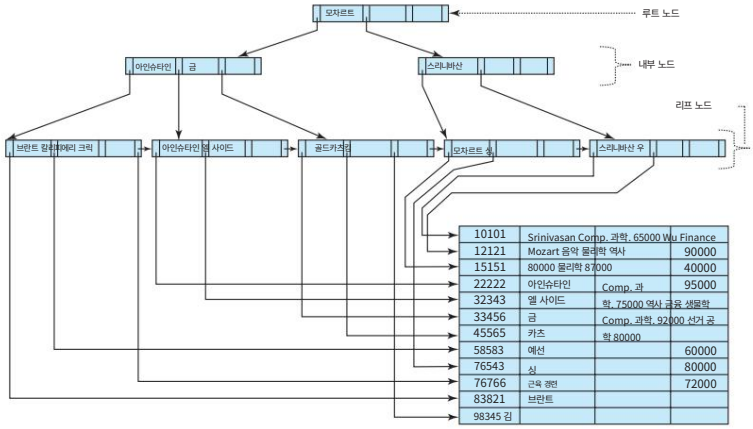
q B+-트리는 다음 속성을 만족하는 루트 트리입니다.

§ 루트에서 리프까지의 모든 경로는 길이가 같습니다.

§ 루트 또는 리프가 아닌 각 노드는 én/2ù 및 n 사이의 자식을 가집니다. § 리프 노드는 é(n-1)/2ù 및 n-1 사이의 값을 가집니다. § 특수한 경우: • 루트가 리프

가 아니면 적어도 2개의 자식이 있습니다. • 루트가 리프

인 경우 (즉, 트리에 다른 노드가 없는 경우) 0과 (n–1) 사이의 값을 가질 수 있습니다.



B+-트리 노드 구조 B+-Tree Node Structure

§ 일반 노드

P1	K1	P2 … Pn-1		Kn-1	Pn

• Ki 는 검색 키 값입니다. • Pi 는 자식에 대한 포인터(비리프 노드의 경우) 또는 레코드 또는 레코드의 버킷에 대한 포인터(리프 노드의 경우)입니다. § 노드의 검색 키는 순서가 지정됩니다.

K1 < K2 < K3 < . . . < Kn–1

(처음에는 중복 키가 없다고 가정하고 나중에 중복 주소를 지정합니다.)

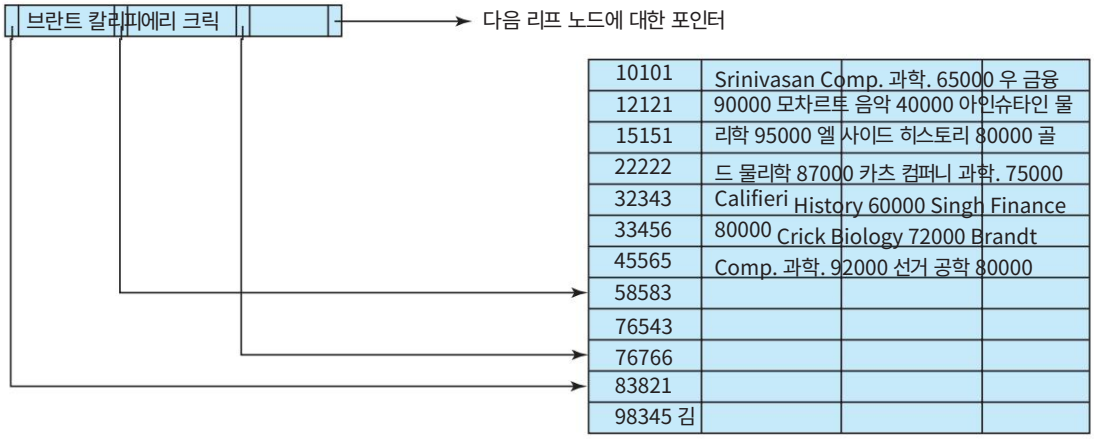
B+-트리의 리프 노드

리프 노드의 속성:

§ i = 1, 2, . . ., n–1, 포인터 Pi는 검색 키 값이 있는 파일 레코드를 가리킵니다.
에 _

§ Li , Lj 가 리프 노드이고 i < j 인 경우 Li 의 검색 키 값은 Lj 의 검색 키 값 보다 작거나 같습니다.

§ Pn은 검색 키 순서 리프 노드 에서 다음 리프 노드를 가리
킵니다.

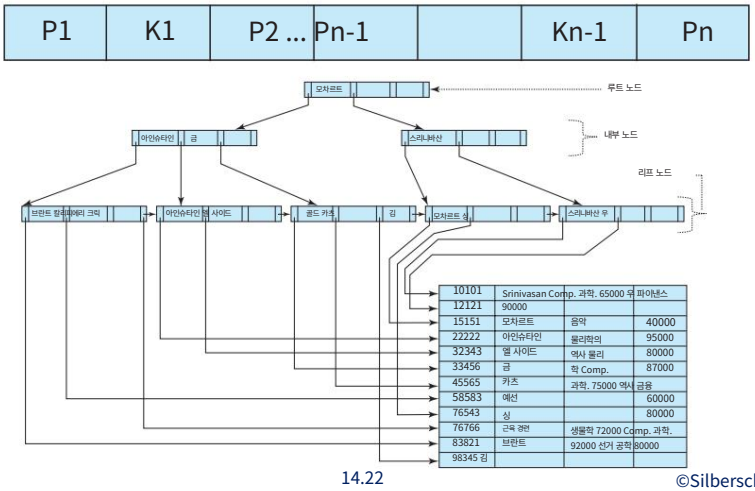


B+-트리의 비리프 노드

§ 비 리프 노드는 리프 노드에서 다단계 희소 인덱스를 형성합니다. m 포인터가 있는 리프 노드가 아닌 경
우:

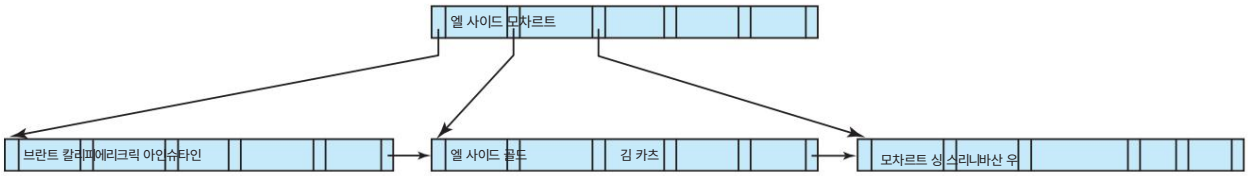
• P1이 가리키는 하위 트리의 모든 검색 키는 K1 보다 작습니다. • 2 £ i £ n – 1 의 경우 Pi
가 가리키는 하위 트리의 모든 검색 키는 Ki–1 이상의 값을 가지며 Ki 보다 작음 • Pn이 가
리키는 하위 트리의 모든 검색 키는 Kn-1 보다 크거나 같은 값을 가집니
다.

• 일반 구조



B+-트리의 예

§ 강사 파일용 B+-트리(n = 6)



§ 리프 노드에는 3~5개의 값이 있어야 합니다.
(é(n–1)/2ù 및 n –1, n = 6).

§ 루트 이외의 비리프 노드는 3~6개 사이여야 함
자녀(é(n/2ù 및 n with n =6).

§ 루트에는 적어도 2명의 자식이 있어야 합니다.

Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
Special cases:
- If the root is not a leaf, it has at least 2 children.
- If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

B+-트리 에 대한 관찰

§ 노드 간 연결은 포인터에 의해 이루어지므로 "논리적으로" 닫힙니다.
블록은 "물리적으로" 가까이 있을 필요가 없습니다.

§ B+-트리 의 리프가 아닌 수준은 희소 인덱스의 계층 구조를 형성합니다. § B +-트리는 상대적으로 적은 수의 수준을 포함합니다. § 루트 아래 수준은 최소 2* én/2ù 값을 가집니다. § 다음 수준은 최소 2* én/2ù * én/2ù 값을 가집니다.

§ .. 등.

- 파일에 K 개의 검색 키 값이 있는 경우 트리 높이는 é logén/2ù(K)ù 이하 이므로 효율적으로 검색할 수 있습니다.

§ 색인이 대수 시간으로 재구성될 수 있으므로 기본 파일에 대한 삽입 및 삭제를 효율적으로 처리할 수 있습니다(앞으로 살펴보겠습니다).

Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
Special cases:
- If the root is not a leaf, it has at least 2 children.
- If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

B+-트리 에 대한 쿼리 Queries on B+-Trees

함수 찾기(v)

1. C=루트 2.

while (C는 리프 노드가 아님)

1. i를 최소 숫자 st v 2 라고 합니다 . 그러한 £ 에 .

숫자 i가 없으면 다음

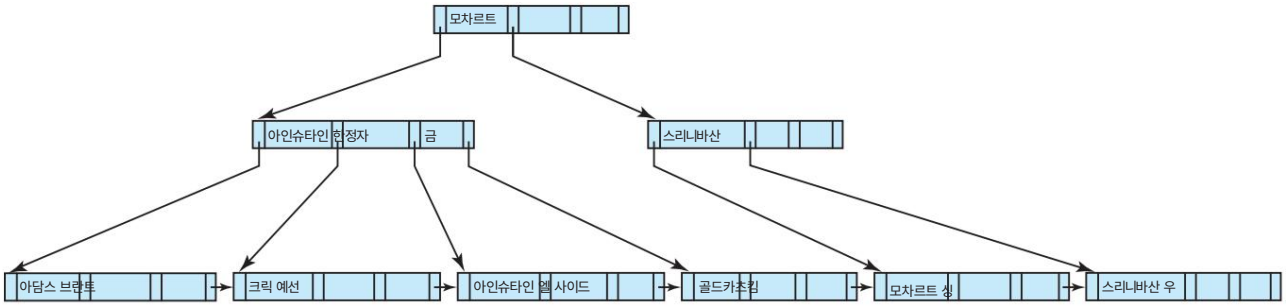
삼. C = C에서 널이 아닌 마지막 포인터 설정 4.

else if (v = C.Ki) 설정 C = Pi +1 5. 그렇지 않

으면 C = C.Pi 설정 3. 일부 i에

대해 Ki = v 이면 C를 반환합니다. Pi 4. else return

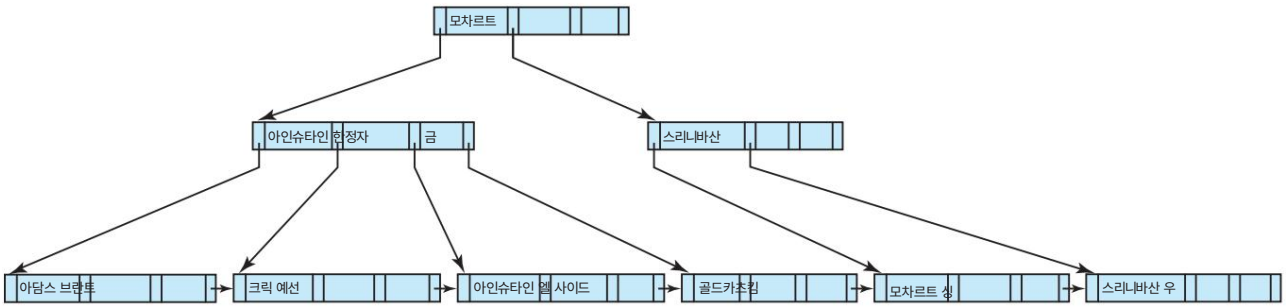
null /* 검색 키 값 v가 있는 레코드가 없습니다. */



B+-트리 에 대한 쿼리 (계속) Queries on B+-Trees (Cont.)

§ 범위 쿼리는 주어진 범위에서 검색 키 값이 있는 모든 레코드를 찾습니다.

- 이러한 모든 레코드 집합을 반환하는 함수 findRange(lb, ub) 에 대한 자세한 내용은 책을 참조하십시오.
- 실제 구현에서는 일반적으로 next() 함수를 사용하여 일치하는 레코드를 한 번에 하나씩 가져오는 반복자 인터페이스를 제공합니다.



B+- 트리에 대한 쿼리 (계속)

§ 파일에 K 개의 검색 키 값이 있는 경우 트리의 높이는 없음
élogén/2ù(K)ù 이상 .

§ 노드는 일반적으로 디스크 블록과 크기가 같으며 일반적으로 4KB이고
n은 일반적으로 인덱스 항목당

약 100 § (4*103) /100 = 40바이트입니다.

§ 검색 키 값이 1백만이고 n = 100인 경우 • 최대

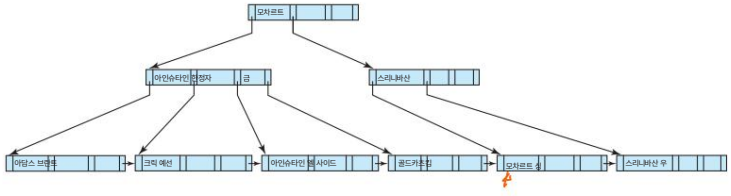
log50(1,000,000) = 4개의 노드가 루트에서 리프까지의 조회 순회에서 액세스됩니다.

§ 100만 개의 검색 키 값이 있는 균형 잡힌 이진 트리와 비교하여 조회 시 약 20개의 노드에 액세스합니다.

• 모든 노드 액세스에는
디스크 I/O, 비용은 약 20밀리초

B+-트리 에 대한 범위 쿼리

```
function findRange(lb, ub)
/* Returns all records with search key value V such that lb ≤ V ≤ ub. */
    Set resultSet = {};
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that lb ≤ C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m
        end
        else if (lb = C.K_i) then Set C = C.P_{i+1}
        else Set C = C.P_i /* lb < C.K_i */
    end
    /* C is a leaf node */
    Let i be the least value such that K_i ≥ lb
    if there is no such i
        then Set i = 1 + number of keys in C; /* To force move to next leaf */
    Set done = false;
    while (not done) begin
        Let n = number of keys in C.
        if ( i ≤ n and C.K_i ≤ ub) then begin
            Add C.P_i to resultSet
            Set i = i + 1
        end
        else if (i ≤ n and C.K_i > ub)
            then Set done = true;
        else if (i > n and C.P_{n+1} is not null)
            then Set C = C.P_{n+1}, and i = 1 /* Move to next leaf */
        else Set done = true; /* No more leaves to the right */
    end
    return resultSet;
```



고유하지 않은 키

§ 검색 키 ai가 고유하지 않은 경우 대신 고유한 복합 키(ai , Ap) 에 인덱스를 생성합니다.
Ap 는 기본 키, 레코드 ID 또는 다
른 속성일 수 있습니다.
고유성을 보장 § ai = v 에
대한 검색은 범위가 (v, - ∞)에서 (v, + ∞) 인 복합 키에 대한 범위 검색으로 구현될 수 있습니다.

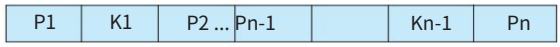
§ 그러나 실제 레코드를 가져오려면 더 많은 I/O 작업이 필요합니다.
- 인덱스가 클러스터링이면 모든 액세스가 순차적입니다.
- 인덱스가 클러스터링되지 않은 경우 각 레코드 액세스에 I/O 작업이 필요할 수 있습니다.

B+-트리에 대한 업데이트 : 삽입

레코드가 이미 파일에 추가되었다고 가정합니다. 하자 !
pr은 레코드에 대한 포인터이고 ! v 레코드의
검색 키 값

1. 검색 키 값이 나타날 리프 노드를 찾습니다.
   1. 리프 노드에 공간이 있으면 리프 노드에 (v, pr) 쌍을 삽입합니다. 2. 그렇지 않으면
   노드를 분할합니다(새 (v, pr) 항목과 함께).
   다음 슬라이드에서 논의하고 상위 노드에 업데이트를 전파합니다.

B+-트리 업데이트 : 삽입(계속)

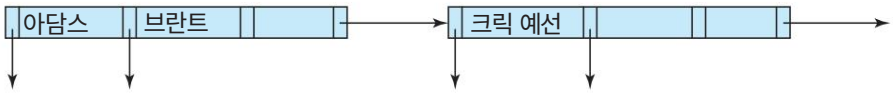


§ 리프 노드 분할:

- n(검색 키 값, 포인터) 쌍을 취합니다(원하는 것을 포함하여).
  삽입) 정렬된 순서로. 첫 번째 én/2ù를 원래 노드에 배치하고
  나머지는 새 노드에 있습니다.
- 새 노드를 p라고 하고 k 를 p 에서 가장 작은 키 값이라고 합니다 . 분할되는 노드의 부모
  에 (k,p)를 삽입합니다.
- 부모가 가득 차 있으면 분할하고 분할을 더 전파합니다 .

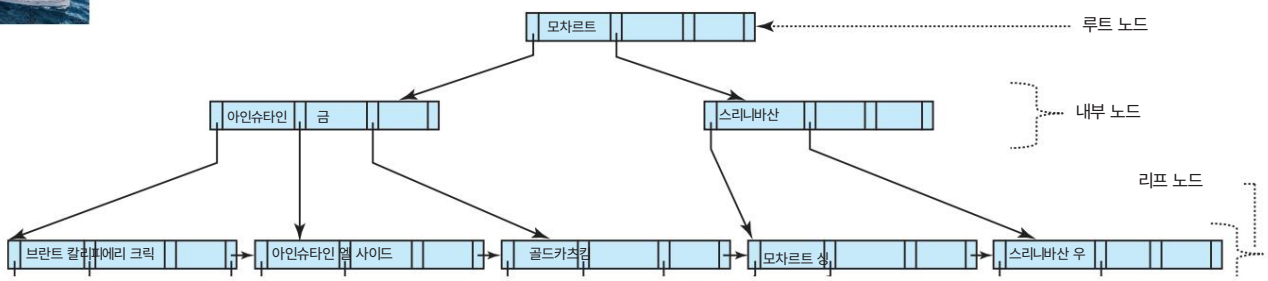
§ 노드 분할은 가득 차지 않은 노드를 찾을 때까지 위쪽으로 진행됩니다.

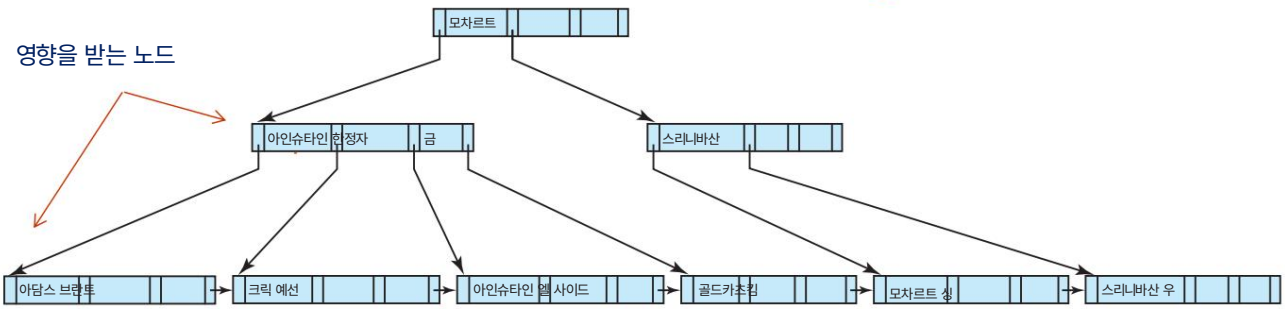
- 최악의 경우 루트 노드가 분할되어 높이가 증가할 수 있습니다.
  나무를 1로.



Adams 삽입 시 Brandt, Califieri 및 Crick이 포함된 노드 분할 결과
다음 단계: (Califieri, pointer-to-new-node) 항목을 부모에 삽입

B+-트리 삽입



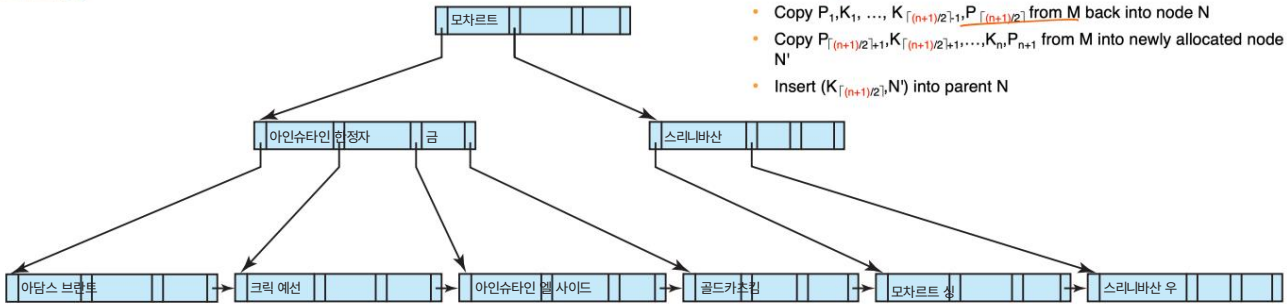


"Adams" 삽입 전후의 B+-트리

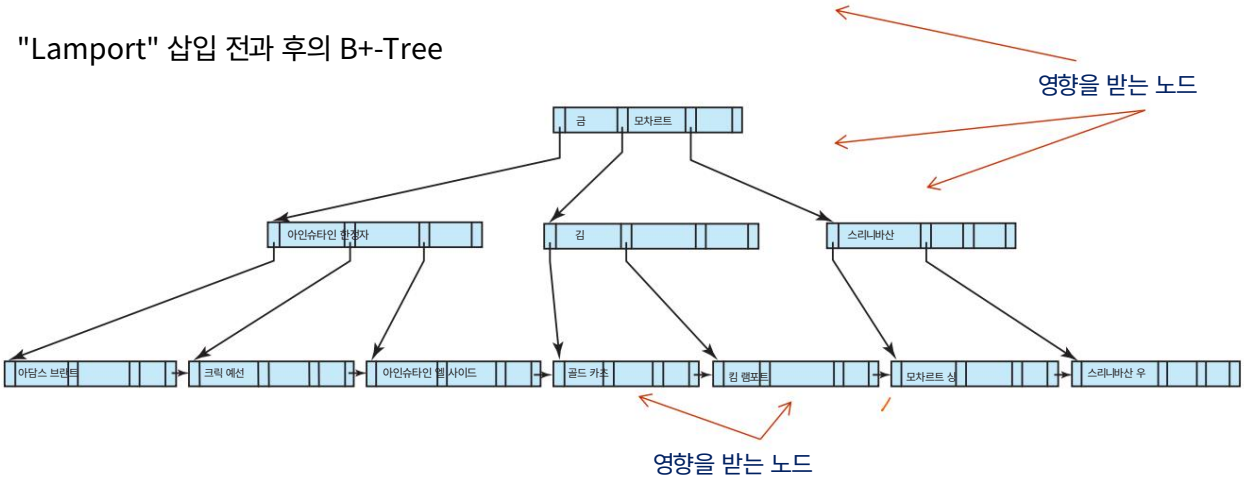
B+-트리 삽입

Splitting a non-leaf node: when inserting (k,p) into an already full internal node N

- Copy N to an in-memory area M with space for n+1 pointers and n keys
- Insert (k,p) into M
- Copy $P_1, K_1, \ldots, K_{\lceil (n+1)/2 \rceil - 1}, P_{\lceil (n+1)/2 \rceil}$ from M back into node N
- Copy $P_{\lceil (n+1)/2 \rceil}, K_{\lceil (n+1)/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
- Insert $(K_{\lceil (n+1)/2 \rceil}, N')$ into parent N



"Lamport" 삽입 전과 후의 B+-Tree

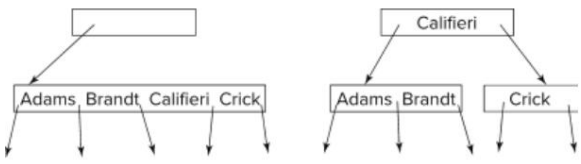


B+-트리 에 삽입 (계속)

§ 리프가 아닌 노드 분할: 이미 가득 찬 내부 노드에 (k,p)를 삽입할 때 노드 N

- n+1 포인터와 n을 위한 공간이 있는 메모리 내 영역 M에 N을 복사합니다. 키 •
- (k,p)를 M에 삽입 •
- P1 ,K1, …, K é(n+1)/2ù-1,P é(n+1)/2ù를 M에서 다시 노드 N으로 복사 •
- Pé(n 복사 +1)/2ù+1,K é(n+1)/2ù+1,…,Kn,Pn+1 M에서 새로 할당된 노드로 N'
- (K é(n+1)/2ù,N')를 부모 N에 삽입

§ 예



§ 책에서 유사 코드 읽기!

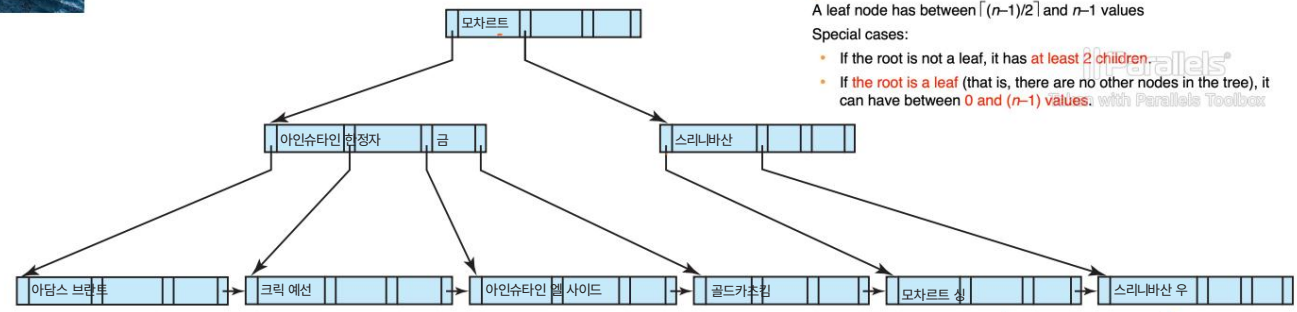
B+-트리 삭제 의 예

Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.

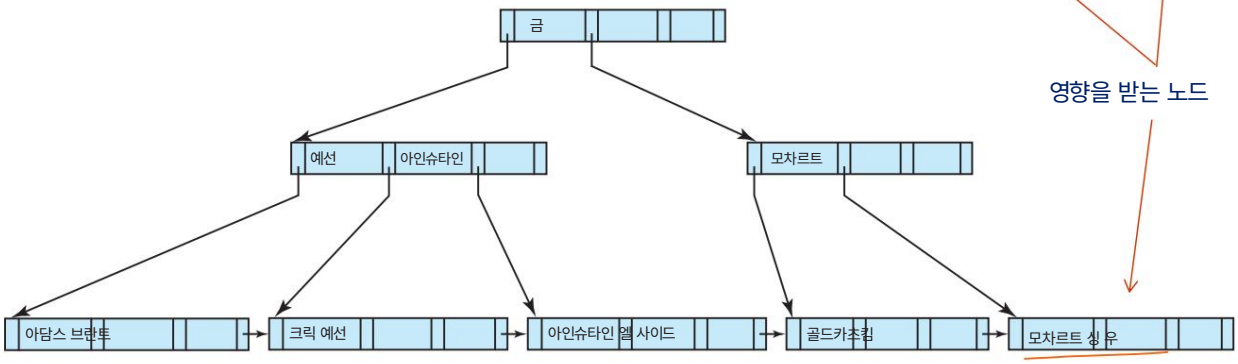
A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values

Special cases:

- If the root is not a leaf, it has at least 2 children.
- If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.



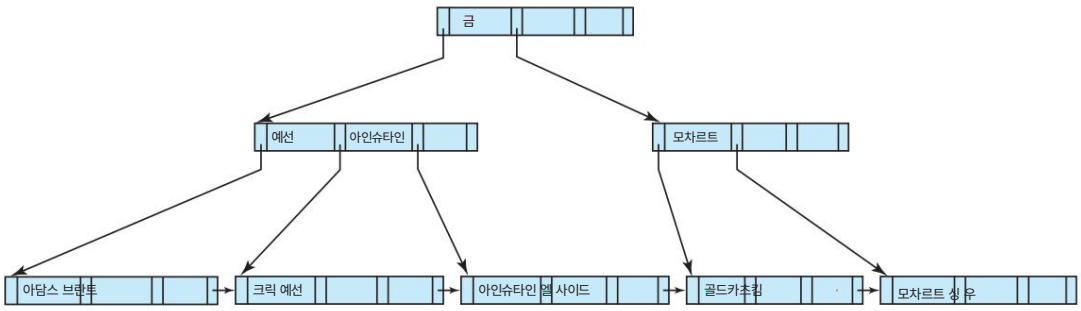
"Srinivasan" 삭제 전과 후



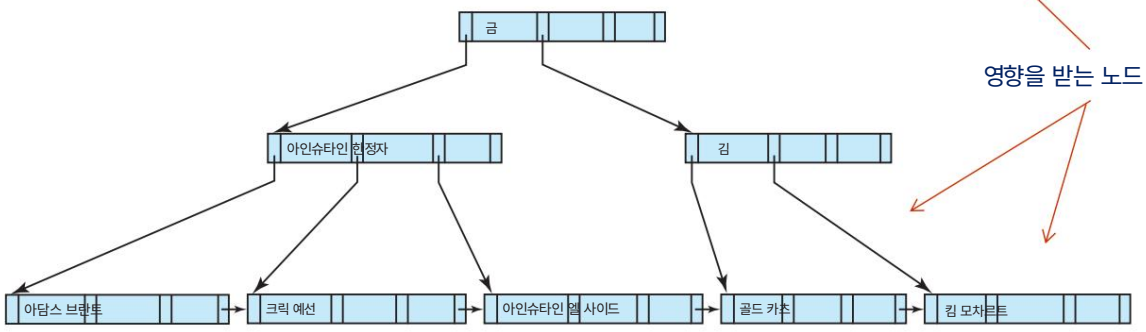
영향을 받는 노드

§ "Srinivasan"을 삭제하면 덜 채워진 잎이 병합 됩니다.

B+-트리 삭제 의 예 (계속)



"Singh" 과 "Wu" 삭제 전과 후

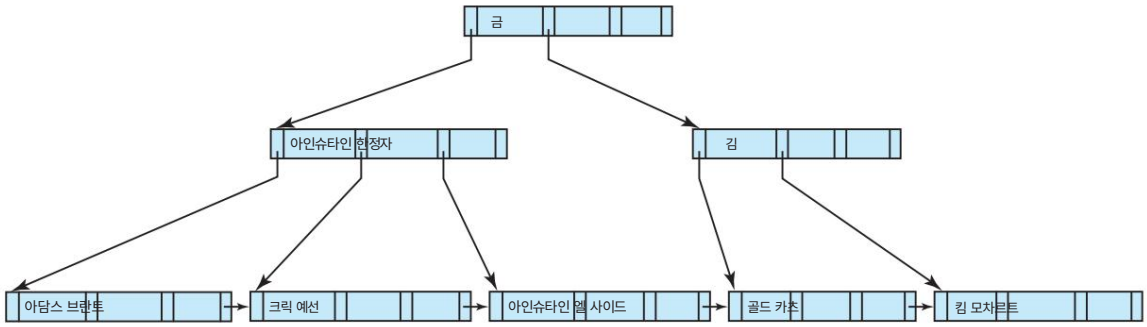


영향을 받는 노드

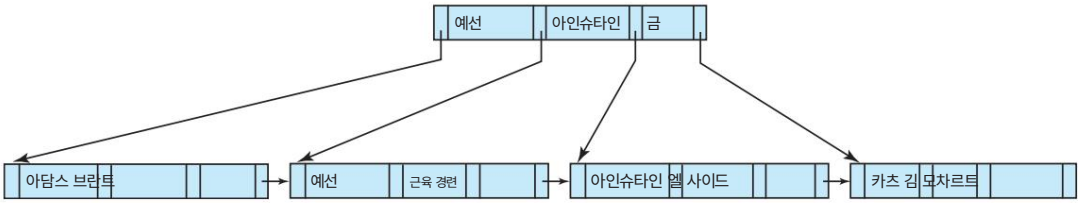
§ Singh과 Wu를 포함하는 리프가 언더풀이 되어 값을 빌렸습니다.
왼쪽 형제의 김

§ 결과적으로 상위 변경의 검색 키 값

B+-트리 삭제의 예 (계속)



"골드" 삭제 전과 후



§ Gold와 Katz가 있는 노드가 underfull이 되어 형제 노드와 병합됨 § 부모 노드가 underfull이 되어 형제 노드와 병합됨

- 두 노드(부모)를 분리하는 값은 병합 시 풀다운됩니다.

§ 그러면 루트 노드에는 자식이 하나만 있고 삭제됩니다.

B+-트리 업데이트 : 삭제

파일에서 이미 삭제된 레코드를 가정합니다. V는 레코드의 검색 키 값이고 Pr은 레코드에 대한 포인터입니다. § 리프 노드에서 (Pr, V) 제거

§ 제거로 인해 노드에 항목이 너무 적고 노드 및 형제의 항목이 단일 노드에 맞는 경우 형제를 병합합니다.

- 두 노드의 검색키 값을 모두 하나의 노드에 삽입 (왼쪽에 있는 것), 다른 노드를 삭제합니다.
- 위의 절차를 반복적으로 사용하여 쌍 (Ki–1, Pi)(Pi가 삭제된 노드에 대한 포인터)을 부모에서 삭제합니다.

B+-트리 업데이트 : 삭제

§ 그렇지 않으면 제거로 인해 노드에 항목이 너무 적지만 노드 및 형제의 항목이 단일 노드에 맞지 않으면 포인터를 재배포합니다.

- 둘 다 최소 항목 수보다 많도록 노드와 형제 사이에 포인터를 재분배합니다.
- 노드의 부모에서 해당 검색 키 값을 업데이트합니다.

§ 노드 삭제는 én/2ù 이상의 포인터가 있는 노드가 발견될 때까지 위쪽으로 캐스케이드될 수 있습니다.

§ 삭제 후 루트 노드에 포인터가 하나만 있으면 삭제되고 유일한 자식이 루트가 됩니다.

업데이트의 복잡성

§ 트리 높이에 비례하는 단일 항목의 삽입 및 삭제 비용(I/O 작업 수 측면에서)

- K개의 항목과 n의 최대 팬아웃을 사용하면 최악의 경우 복잡도는 다음과 같습니다.
  항목의 삽입/삭제는 O(logén/2ù(K)) § 실제로는 I/

O 작업 수가 더 적습니다.

- 내부 노드는 버퍼에 있는 경향이 있습니다.
- 분할/병합은 드물며 대부분의 삽입/삭제 작업은 리프에만 영향을 미칩니다.
  마디

§ 평균 노드 점유율은 삽입 순서에 따라 다름

- ! – 임의로, 정렬된 순서로 삽입된 ½
  "

고유하지 않은 검색 키

§ 앞에서 설명한 계획에 대한 대안

• 별도의 블록에 있는 버킷(나쁜 생각) • 각 키가 있는 튜

플 포인터 목록 § 긴 목록을 처리하기 위한 추가 코

드 § 검색 키에 중복 항목이 많으면 튜플 삭

제 비용이 많이 들 수 있음(이유?) • 최악의 경우 복잡성은 선의!

§ 낮은 공간 오버헤드, 쿼리에 대한 추가 비용 없음 • 레코드 식별자

를 추가하여 검색 키를 고유하게 만듭니다.

§ 키에 대한 추가 스토리지 오버헤드 § 삽입/삭

제를 위한 간단한 코드 § 널리 사용됨

B+-트리 파일 구성

§ B+-트리 파일 구성: • B+-트리 파일 구

성 의 리프 노드는 대신 레코드를 저장합니다.
포인터

• 데이터 레코드가 있는 경우에도 클러스터된 데이터 레코드를 유지하는 데 도움이 됩니다.
삽입/삭제/업데이트

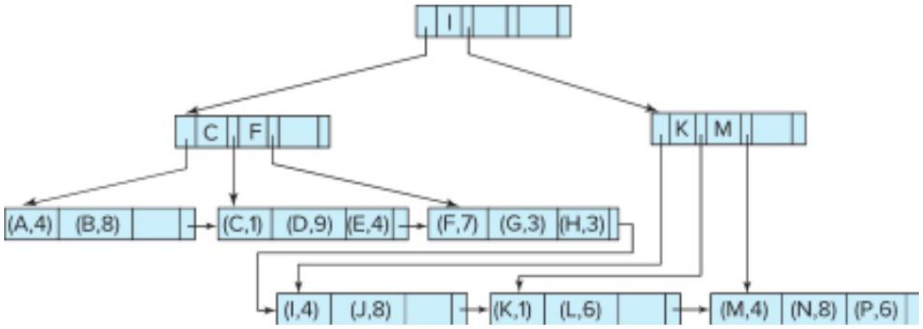
§ 리프 노드는 여전히 반만 채워야 합니다.

• 레코드가 포인터보다 크기 때문에 최대
리프 노드에 저장할 수 있는 레코드는 비리프 노드의 포인터 수보다 적습니다.

§ 삽입 및 삭제는 삽입 및 삭제와 동일한 방식으로 처리됩니다.
B+ 트리 인덱스 의 항목 삭제 .

B+-트리 파일 구성(계속)

§ B+-트리 파일 구성의 예



§ 레코드가 레코드보다 더 많은 공간을 사용하므로 좋은 공간 활용이 중요합니다.
포인터.

§ 공간 활용도를 높이기 위해 재분배에 더 많은 형제 노드 참여
분할 및 병합 중 • 재분배에

2개의 형제를 포함하면(가능한 경우 분할/병합을 피하기 위해) 각 노드가 최소 ë2n / 3û 를 갖게 됩니다. 항목

인덱싱의 기타 문제

§ 레코드 재배치 및 보조 인덱스

- 레코드가 이동하면 레코드 포인터를 저장하는 모든 보조 인덱스를 업데이트 해야 합니다.

• B+-트리 파일 조직 의 노드 분할은 매우 비쌉니다. • 솔루션: 레코드 대신

B+-트리 파일 조직 의 검색 키를 사용합니다.
보조 인덱스의 포인터

§ B+-트리 파일 구성 검색 키가 고유하지 않은 경우 레코드 ID 추가 § 레코드를 찾기 위한 파일 구성 추가 순회

• 쿼리 비용이 높지만 노드 분할이 저렴합니다.

인덱싱 문자열

§ 키로서의 가변 길이 문자열
- 가변 팬아웃
- 포인터의 개수가 아닌 공간 활용도를 기준으로 분할

§ 접두사 압축
- 내부 노드의 키 값은 전체 키의 접두사가 될 수 있습니다.
  § 키 값으로 구분된 하위 트리의 항목을 구별할 수 있도록 충분한 문자 유지
  - 예: "Silas"와 "Silberschatz"는 "Silb"로 구분할 수 있습니다. • 공통 접두사를 공유

하여 리프 노드의 키를 압축할 수 있습니다.

대량 로딩 및 상향식 빌드

§ 항목을 한 번에 하나씩 B+-트리에 삽입하려면 항목당 [3] 1 IO가 필요합니다.
- 리프 수준이 메모리에 맞지 않는다고 가정하는 경우 • 한 번에

많은 수의 항목을 로드하는 데 매우 비효율적일 수 있습니다 (대량 로드).

§ 효율적인 대안 1:
- 먼저 항목을 정렬합니다(섹션 12.4에서 나중에 설명하는 효율적인 외부 메모리 정렬 알고리즘 사용).
- 정렬된 순서로 삽입
  § 삽입은 기존 페이지로 이동(또는 분할 유발) § 훨씬 향상된 IO 성능이

지만 대부분의 리프 노드는 절반만 찼습니다.

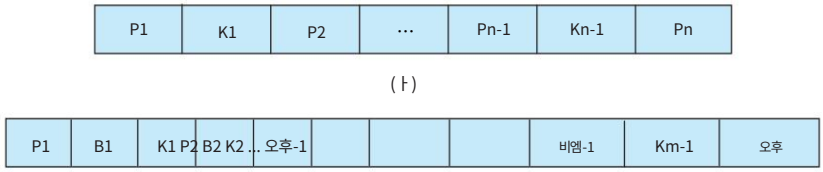
§ 효율적인 대안 2: 상향식 B+-트리 구성
- 이전과 마찬가지로 항목 정렬
- 그런 다음 리프 레벨부터 시작하여 레이어별로 트리를 생성합니다.
  § 운동으로 세부 사항
- 대부분의 데이터베이스 시스템에서 대량 로드 유틸리티의 일부로 구현

B-트리 인덱스 파일 B-Tree Index Files

§ B+-트리 와 유사 하지만 B-트리는 검색 키 값이 한 번만 나타날 수 있도록 허용합니다.
검색 키의 중복 저장을 제거합니다.

§ 리프가 아닌 노드의 검색 키는 B-트리의 어디에도 나타나지 않습니다. 리프가 아닌 노드의 각 검색 키에 대한 추가 포인터 필드가 포함 되어야 합니다. (비)

§ 일반화된 B-트리 리프 노드 (a)



§ Nonleaf 노드 – 포인터 Bi는 버킷 또는 파일 레코드 포인터입니다.

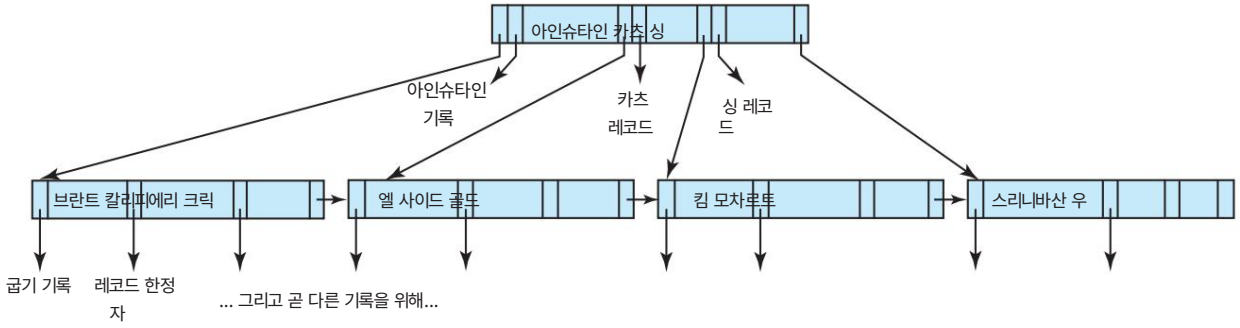
B-Tree 인덱스 파일(계속) B-Tree Index Files (Cont.)

§ B-Tree 인덱스의 장점: • 해당 B+-Tree보다
적은 트리 노드를 사용할 수 있습니다. • 때때로 리프에 도달하기 전에 검색 키 값을
찾을 수 있음
마디.

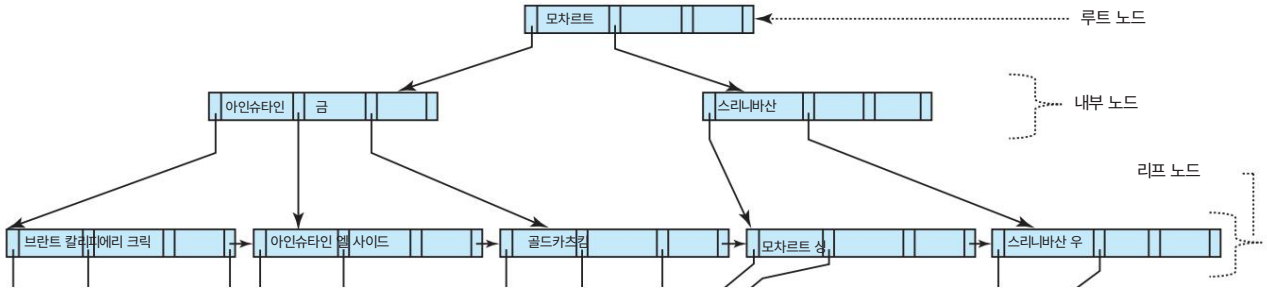
§ B-Tree 인덱스의 단점:
• 모든 검색 키 값 중 작은 부분만 초기에 발견됩니다 . • 리프가 아닌 노드가 더 크기 때문
에 팬아웃이 줄어듭니다. 따라서 B-Tree는 일반적으로 해당 B+-Tree 보다 깊이가 깊습니다 . •
B+-Tree 보다 삽입 및 삭제가 더 복잡합니다. • 구현이 B+-Tree 보다 어렵습니
다 .

§ 일반적으로 B-Tree의 장점이 단점보다 크지 않습니다.

B-Tree 인덱스 파일 예제



동일한 데이터에 대한 B-트리(위) 및 B+-트리(아래)



플래시에서 인덱싱

§ 랜덤 I/O 비용은 플래시에서 훨씬 낮음

- 읽기/쓰기에 20~100마이크로초

§ 쓰기가 제자리에 있지 않으며 (결국) 더 많은 비용이 드는 지우기가 필요합니다. § 최적 의 페이지 크기 는 훨씬 더 작습니다 . 플래시 최적화 검색 트리를 위한 페이지 쓰기

메인 메모리의 인덱싱

§ 메모리의 임의 액세스 • 디스크/플래시보
다 훨씬 저렴합니다 . 그러나 캐시 읽기에 비해 여전
히 비쌉니다. • 캐시를 최대한 활용하는 데이터 구조가 바람직합니
다. • 큰 B+-트리 노드 내에서 키 값에 대한 이진 검색으로 많은 캐시 누락이 발생합
니다.

§ 캐시 라인에 맞는 작은 노드가 있는 B+- 트리가 감소하는 것이 좋습니다.
캐시 미스

§ 핵심 아이디어: 큰 노드 크기를 사용하여 디스크 액세스를 최적화하지만 배열을 사용하
는 대신 노드 크기가 작은 트리를 사용하여 노드 내에서 데이터를 구조화합니다.

해싱

정적 해싱

§ 버킷 은 하나 이상의 항목을 포함하는 저장 단위입니다(버킷
일반적으로 디스크 블록). • 해시
함수를 사용하여 검색 키 값에서 항목의 버킷을 얻습니다.

§ 해시 함수 h는 모든 검색 키 값 집합 K에서 모든 버킷 주소 집합 B까지의 함수입니다.

§ 해시 함수는 액세스, 삽입 및
삭제.

§ 서로 다른 검색 키 값을 가진 항목은 동일한 항목에 매핑될 수 있습니다.
버킷; 따라서 항목을 찾으려면 전체 버킷을 순차적으로 검색해야 합니다.

§ 해시 인덱스 에서 버킷은 레코드에 대한 포인터가 있는 항목을 저장합니
다. § 해시 파일 구성 버킷 에서는 레코드를 저장합니다.

버킷 오버플로 처리

§ 버킷 오버플로우는 다음으로 인해 발생할 수 있습니다.

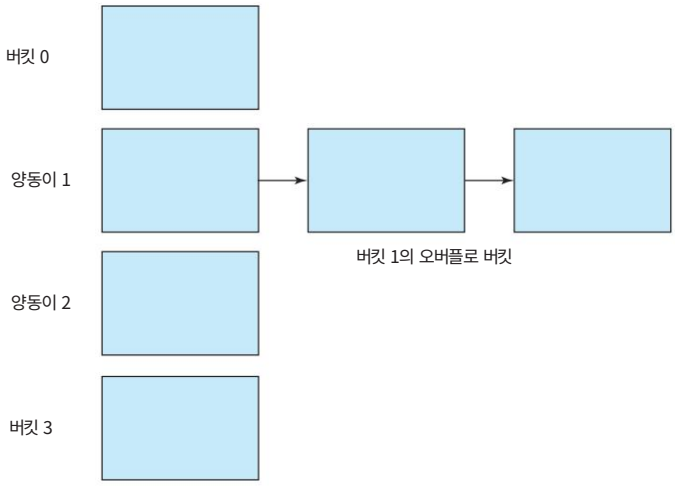
- 불충분한 버킷
- 기록 배포의 왜곡. 이는 다음 두 가지 이유로 발생할 수 있습니다.
  - § 여러 레코드가 동일한 검색 키 값을 가짐 § 선택
    된 해시 함수가 키의 균일하지 않은 분포를 생성함
    값

§ 버킷 오버플로 가능성을 줄일 수는 있지만 제거할 수는 없습니다. 오버플로 버킷을 사용하여 처리됩니다 .

버킷 오버플로 처리(계속) Handling of Bucket Overflows (Cont.)

§ 오버플로 연결 - 주어진 버킷의 오버플로 버킷이 연결된 목록에서 함께 연결됩니다.

§ 위의 방식을 폐쇄 주소 지정 (폐쇄 해싱 이라고도 함)
또는 사용하는 책에 따라 개방형 해싱) • 흐름 버킷을 사용하지 않는 개방형 주소 지정 (사용하는 책에 따라 개방형 해싱 또는 폐쇄형 해싱 이라고도 함)이라는 대안은 데이터베이스 응용 프로그램에 적합하지 않습니다.



해시 파일 구성의 예 Example of Hash File Organization

dept_name을 키로 사용하는 강사 파일의 해시 파일 구성(다음 슬라이드의 그림 참조)

§ 10개의 버킷이 있으며,

§ I 번째 문자 의 이진 표현은 정수 나.

§ 해시 함수는 이진 표현의 합을 반환합니다.
문자 모듈로 10

• 예 h(음악) = 1 h(역사) = 2 h(물리) = 3 h(전자공학) = 3

해시 파일 구성의 예 Example of Hash File Organization

dept_name을 키로 사용하여 강사 파일의 해시 파일 구성.

버킷 0

양동이 1

15151 모차르트 음악 40000			

양동이 2

32343	엘 사이드	역사	80000
58583	예선	역사	60000

버킷 3

22222	아인슈타인	물리학 물리	95000
33456	골드	학 전기 공	87000
98345	김	학 80000	

버킷 4

12121	우	금융 90000	
76543 싱		재원	80000

버킷 5

76766 크릭		생물학 72000	

버킷 6

10101 스리니바산 콤프. 과학. 65000			
45565 비교 과학. 75000 Katz 83821			
Brandt Comp. 과학.			92000

버킷 7

정적 해싱의 결함 Deficiencies of Static Hashing

§ 정적 해싱에서 함수 h는 검색 키 값을 버킷 주소의 고정 집합 B에 매핑합니다. 데이터베이스는 시간이 지남에 따라 커지거나 줄어듭니다.

- 초기 버킷 수가 너무 적고 파일이 커지면 너무 많은 오버플로로 인해 성능이 저하됩니다.
- 예상되는 성장을 위해 공간이 할당되면 초기에 상당한 양의 공간이 낭비될 것입니다(그리고 버킷이 가득 차게 됩니다).
- 데이터베이스가 축소되면 다시 공간이 낭비됩니다. § 하나의

솔루션: 새로운 해시 함수로 파일을 주기적으로 재구성

- 비용이 많이 들고 정상 운영 방해

§ 더 나은 솔루션: 버킷 수를 동적으로 수정할 수 있습니다.

동적 해싱

§ 주기적인 재해싱
- 해시 테이블의 항목 수가 해시 테이블 크기의 (예를 들어) 1.5배가 되면,
  - § 크기의 2배 크기의 새 해시 테이블 생성
    이전 해시 테이블
  - § 모든 항목을 새 테이블로 다시 해시

§ 선형 해싱 • 증분
방식으로 다시 해싱 수행

§ 확장 가능한 해싱
- 여러 해시가 버킷을 공유하는 디스크 기반 해싱에 맞춤화됨
  값
- 버킷 수를 두 배로 늘리지 않고 해시 테이블의 항목 수를 두 배로 늘림

정렬된 인덱싱과 해싱의 비교

§ 주기적인 재구성 비용 § 삽입 및 삭
제의 상대적 빈도 § 최악의 비용으로 평균 액세스 시간
을 최적화하는 것이 바람직합니까?
케이스 액세스 시간?

§ 예상 쿼리 유형: • 해싱은 일반
적으로 지정된 레코드를 검색하는 데 더 좋습니다.
키의 값.
- 범위 쿼리가 일반적인 경우 정렬된 인덱스가 선호됩니다.

§ 실제로:
- PostgreSQL은 해시 인덱스를 지원하지만 성능이 좋지 않아 사용을 권장하지 않습니다.
  성능 • Oracle
  은 정적 해시 조직을 지원하지만 해시 인덱스는 지원하지 않습니다. • SQLServer
  는 B+-트리 만 지원합니다.

다중 키 액세스 Multiple Key Access

§ 특정 유형의 쿼리에 대해 여러 인덱스를 사용합니다. § 예:

아이디 선택

강사 로부터

여기서 dept_name = "재무" 및 급여 = 80000

§ 단일 속성에 대한 인덱스를 사용하여 쿼리를 처리하기 위한 가능한 전략: 1. dept_name에서 인덱스를 사용하여 부서 이름이 Finance인 강사를 찾습니다. 시험 급여 = 80000 2. 급여에 대한 인덱스를 사용하여 급여가 $80000인 강사를 찾으십시오. 시험 dept_name = "재무".

3. dept_name 인덱스를 사용하여 "재무" 부서와 관련된 모든 레코드에 대한 포인터를 찾습니다. 마찬가지로 급여에 인덱스를 사용합니다. 얻은 두 포인터 세트의 교차점을 가져옵니다.

여러 키의 인덱스 Indices on Multiple Keys

§ 복합 검색 키는 둘 이상의 검색 키를 포함하는 검색 키입니다. 기인하다

- 예, (dept_name, salary) § 사전

식 순서: (a1, a2) < (b1, b2) if 둘 중 하나

- a1 < b1 또는 •

a1=b1 및 a2 < b2

여러 속성에 대한 인덱스 Indexing Multiple Attributes

결합된 검색 키(dept_name, salary)에 대한 인덱스가 있다고 가정
합니다.

§ where 절 사용

여기서 dept_name = "재무" 및 급여 = 80000
(dept_name, salary)의 인덱스는 두 조건을 모두 만족하는 레코드만 가져오는 데 사
용할 수 있습니다. • 덜 효율

적인 별도의 인덱스 사용 — 조건 중 하나만 만족하는 많은 레코드(또는 포인터)
를 가져올 수 있습니다. § 또한 dept_name = "Finance" 이고 급여

< 80000 인 경우 효율적으로 처리할
수 있습니다. § 그러나 dept_name < "Finance" 이고

balance = 80000 인 경우 효율적으
로 처리할 수 없습니다.

• 첫 번째는 만족하지만 두 번째는 만족하지 않는 많은 레코드를 가져올 수 있습니다.
상태

다른 기능들

§ 커버링 인덱스 • (일부)

쿼리가 실제 레코드를 가져오는 것을 피할 수 있도록 인덱스에 추가 속성을 추가합니다.

• 리프에만 추가 속성 저장

§ 왜? §

2차 지수에 특히 유용함 • 왜?

지수 생성

§ 예제 인덱스
생성 take_pk on take (ID,course_ID, 연도, 학기, 섹션) drop index takes_pk

§ 대부분의 데이터베이스 시스템은 인덱스 유형 지정 및 클러스터링을 허용합니다. § 모든 데이터베이스에서 자동으로 생성되는 기본 키에 대한 인덱스

- 왜? § 관계
  에 튜플이 삽입될 때마다 인덱스를 사용하여 기본 키가 제약 조건이 위반되지 않음(즉, 기본 키 값에 중복이 없음)
  § 기본 키에 인덱스가 없으면 튜플이 삽입될 때마다 기본 키 제약 조건이 충족되는지 확인하기 위해 전체 관계를 스캔해야 합니다.

§ 일부 데이터베이스는 외래 키 속성에 대한 인덱스도 생성합니다.

- 이러한 인덱스가 이 쿼리에 유용한 이유는 무엇입니까?
  § take σname='Shankar' (학생)

§ 인덱스는 조회 속도를 크게 높일 수 있지만 업데이트에 비용이 듭니다.

- 쿼리 및 업데이트 워크로드를 기반으로 인덱스를 선택하는 데 도움이 되는 여러 데이터베이스에서 지원되는 인덱스 튜닝 도우미/마법사

SQL의 인덱스 정의

§ 인덱스 생성
<relation-name>(<attribute-list>) 에 인덱스 <index-name> 생성
예: create index b-index on branch(branch_name) § create unique index를 사용하여 조건을 간접적으로 지정하고 시행합니다.
검색 키가 후보 키라는 것
§ SQL 고유 무결성 제약 조건이 지원되는 경우 실제로 필요하지 않음 § 인덱스 삭제
드롭 인덱스 <index-name>
§ 대부분의 데이터베이스 시스템은 인덱스 유형 지정을 허용하고 클러스터링.

최적화된 인덱스 쓰기

§ B+-트리 의 성능은 쓰기 집약적 워크로드에 대해 좋지 않을 수 있습니다.

- 모든 내부 노드가 메모리에 있다고 가정할 때 리프당 I/O 1개
- 자기 디스크 사용, 디스크당 초당 100개 미만 삽입
- 플래시 메모리 사용, 삽입당 1페이지 덮어쓰기

§ 쓰기 비용을 줄이기 위한 두 가지 접근 방식
- 로그 구조 병합 트리
- 버퍼 트리

로그 구조 병합(LSM) 트리

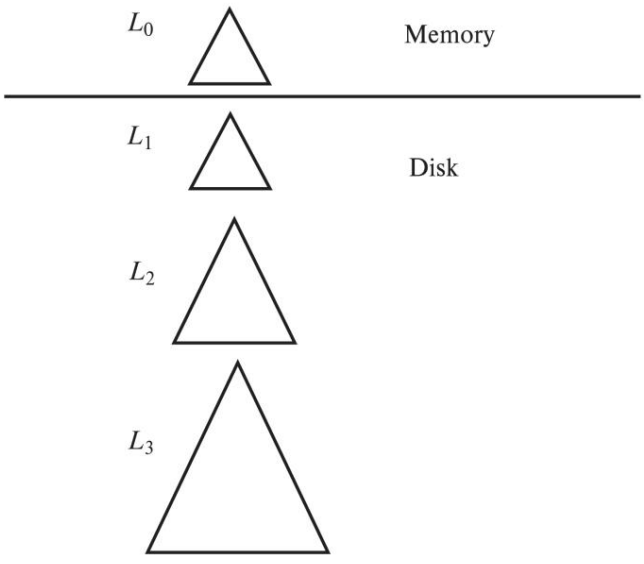
§ 다음에 대한 삽입/쿼리만 고려
지금

§ 에 처음 삽입된 레코드
메모리 트리(L0 트리) § 인

메모리 트리가 가득 차면,
디스크로 이동된 레코드(L1 트리)
- 다음을 사용하여 구성된 B+-트리
  기존 L1 트리와 L0 트리 의 레코드를 병합하여 상향식 빌드 § L1 트리가 일정 임계

값을 초과 하면 L2 트리 로 병합

- 더 많은 레벨을 위한 등등
- Li+1 트리 의 크기 임계값은 다음에 대한 크기 임계값의 k배입니다.
  리 트리



LSM 트리(계속)

§ LSM 접근법의 이점

- 삽입은 순차 I/O 작업만 사용하여 수행됩니다. • 공간 낭비를 방지하기 위해 리프가 가득 찼습니다. • 비교할 때 삽입된 레코드당 I/O 작업 수가 감소했습니다.

일반 B+-트리로(일부 크기까지) § LSM 접근 방식의 단점

- 쿼리는 여러 트리를 검색해야 합니다.
- 각 레벨의 전체 내용을 여러 번 복사

§ 단계별 병합 인덱스 • 각 수준에 여러 트리가 있는 LSM 트리의 변형 • LSM 트리에 비해 쓰기 비용 감소 • 하지만 쿼리 비용이 훨씬 더 비쌉니다.

§ 대부분의 트리에서 조회를 피하기 위한 블룸 필터 § 자세한 내용은 24장에서 다룹니다.

LSM 트리(계속)

§ 특수 "삭제" 항목을 추가하여 삭제 처리

- 조회는 원래 항목과 삭제 항목을 모두 찾고 일치하는 삭제 항목이 없는 항목만 반환해야 합니다.
- 트리가 병합될 때 원본과 일치하는 삭제 항목을 찾으면 입력하면 둘 다 삭제됩니다. §

삽입+삭제를 사용하여 처리되는 업데이트 § 디스크 기반 인덱스에 대해 LSM 트리가 도입되었습니다.

- 그러나 플래시 기반 인덱스로 삭제를 최소화하는 데 유용합니다.
- LSM 트리의 단계별 병합 변형은 많은 BigData에서 사용됩니다. 스토리지 시스템

§ Google BigTable, Apache Cassandra, MongoDB § 최근에는 MySQL의 SQLite4, LevelDB 및 MyRocks 스토리지 엔진

버퍼 트리 Buffer Tree

§ LSM 트리의 대안

§ 핵심 아이디어: B+-트리의 각 내부 노드에는 삽입을 저장할 버퍼가 있습니다.

- 버퍼가 가득 차면 삽입이 더 낮은 수준으로 이동합니다.
- 큰 버퍼를 사용하면 매번 많은 레코드가 하위 레벨로 이동합니다. • 레코드당 I/O는 이에 따라 감소합니다.

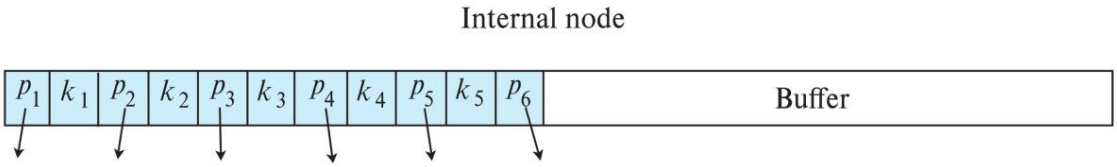
§ 혜택

- 쿼리 오버헤드 감소 • 모든 트리 인덱스 구조

와 함께 사용 가능 • PostgreSQL GiST(일반화 검색 트리) 인덱스에

서 사용

§ 단점: LSM 트리보다 더 많은 랜덤 I/O



비트맵 인덱스 Bitmap Indices

§ 비트맵 인덱스는 여러 키에 대한 효율적인 쿼리를 위해 설계된 특별한 유형의 인덱스입니다.

§ 관계의 레코드는 다음에서 순차적으로 번호가 매겨지는 것으로 가정합니다.
예를 들

어, 0 • 숫자 n이 주어지면 레코드 n을 쉽게 검색할 수 있어야 합니다.

§ 레코드가 고정 크기인 경우 특히 쉬움

§ 상대적으로 적은 수의 개별 속성을 취하는 속성에 적용 가능
값

- 예, 성별, 국가, 주, … • 예, 소득 수준(0-9999,

10000-19999, 20000-50000, 50000-무한대와 같은 소수의 수준으로 분할된 소득)

§ 비트맵은 단순히 비트의 배열입니다.

비트맵 인덱스(계속) (Cont.)

§ 가장 단순한 형태의 속성에 대한 비트맵 인덱스는 속성의 각 값에 대한 비트맵을 가집니다.

- 비트맵은 레코드만큼 많은 비트를 가집니다.
- 값 v에 대한 비트맵에서 레코드에 대한 비트는 레코드가 속성에 대한 값 v를 가지고 있으면 1이고 그렇지 않으면 0입니다.

§ 예

레코드 번호	ID	소득 수준	성별
0	76766	중	L1
1	22222	삼	L2
2	12121	삼	L1
삼	15151	중	L4
4	58583	삼	L3

성별 에 대한 비트맵

중	10010
삼	01101

income_level 에 대한 비트맵

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

비트맵 인덱스(계속) (Cont.)

§ 비트맵 인덱스는 여러 속성에 대한 쿼리에 유용합니다 . • 단일 속성 쿼리에는 특별히 유용하지 않습니다.

§ 질의는 비트맵 연산을 사용하여 응답됨 • 교차(and) • 합집합(or) § 각 연산은 동일한 크기의 두 비트맵을 취하고 해당 비트에 연산을 적용하여 결과 비트맵을 얻음

- 예, 100110 AND 110011 = 100010
  - 100110 또는 110011 = 110111
  - 100110이 아님 = 011001
- 소득 수준이 L1인 남성: 10010 AND 10100 = 10000

§ 그러면 필요한 튜플을 검색할 수 있습니다. § 일치하는 튜플의 수를 세는 것이 훨씬 빠릅니다.

비트맵 인덱스(계속) (Cont.)

§ 비트맵 인덱스는 일반적으로 관계 크기에 비해 매우 작음

- 예, 레코드가 100바이트인 경우 단일 비트맵의 공간은 1/800입니다.
  관계에서 사용하는 공간. § 고유

  한 속성 값의 수가 8인 경우 비트맵은 1%에 불과합니다.
  관계 크기

비트맵 연산의 효율적인 구현

§ 비트맵은 단어로 압축됩니다. 단일 단어 및 (기본 CPU 명령) 계산 및 한 번에 32 또는
64비트

- 예: 31,250개의 명령어로 100만 비트 맵을 편집할 수 있습니다.

§ 트릭으로 1의 수를 빠르게 세는 것이 가능합니다.

- 각 바이트를 사용하여 이진 표현에서 각각 1의 카운트를 저장하는 256개 요소의 미리 계산된 배열로 인덱싱합니다.
  - § 더 높은 메모리에서 속도를 더 높이기 위해 바이트 쌍을 사용할 수 있습니다.
    비용
- 검색된 카운트를 합산합니다.

§ 잎 수준에서 Tuple-ID 목록 대신 비트맵을 사용할 수 있습니다.
일치하는 레코드가 많은 값의 경우 B+-트리

- 레코드의 > 1/64가 튜플을 가정할 때 해당 값을 갖는 경우 가치가 있습니다.
  아이디는 64비트
- 위의 기술은 비트맵과 B+-트리 인덱스 의 이점을 병합합니다.

공간 및 시간 지표

공간 데이터

§ 데이터베이스는 래스터 이미지 외에도 라인, 폴리곤과 같은 데이터 유형을 저장할 수 있습니다.

- 관계형 데이터베이스가 공간 정보를 저장하고 검색할 수 있습니다.
- 쿼리는 공간 조건(예: 포함 또는 겹침)을 사용할 수 있습니다.
- 쿼리는 공간 및 비공간 조건을 혼합할 수 있습니다.

§ 점이나 객체가 주어졌을 때 가장 가까운 이웃 쿼리는 가장 가까운 것을 찾습니다.
주어진 조건을 만족하는 객체.

§ 범위 쿼리는 공간 영역을 처리합니다. 예를 들어, 거짓말하는 물건을 요구
부분적으로 또는 완전히 지정된 지역 내부. § 영역의 교차

점 또는 합집합을 계산하는 쿼리 . § 조인 역할을 하는 위치와 두 공간 관계의 공간

조인
기인하다.

공간 데이터의 인덱싱

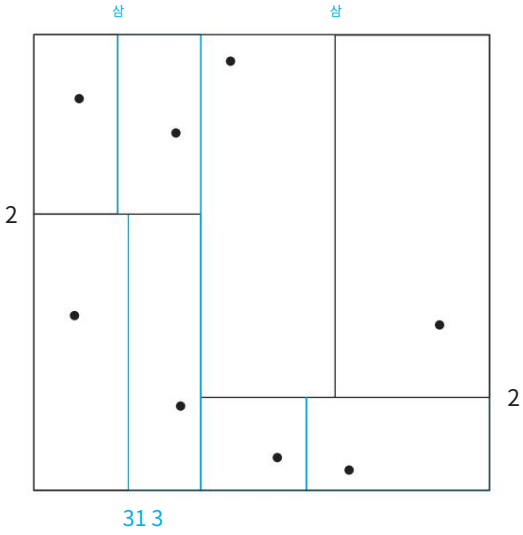
§ kd 트리 - 다차원에서 인덱싱에 사용되는 초기 구조.

§ kd 트리의 각 수준은 공간을 둘로 분할합니다.

- 다음에 대해 하나의 측정기준을 선택합니다.
  트리의 루트 수준에서 분할.
- 차원을 순환하면서 다음 수준 등에서 노드 분할을 위해 다른 차원을 선택합니다.

§ 각 노드에서 하위 트리에 저장된 포인트의 약 절반은 한쪽에 있고 절반은 다른쪽에 있습니다.

§ 노드의 포인트 수가 지정된 수보다 적으면 분할이 중지됩니다.



§ kdB 트리는 kd를 확장합니다.
각 내부 노드에 대해 여러 자식 노드를 허용하는 트리; 보조 스토리지에 적합합니다.

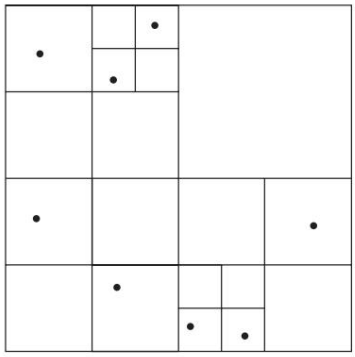
쿼드트리에 의한 공간 분할

§ 쿼드트리의 각 노드는 직사각형 공간 영역과 연관됩니다. 최상위 노드는 전체 대상 공간과 연결됩니다.

§ 각각의 리프 노드가 아닌 노드는 해당 영역을 4개의 동일한 크기의 사분면으로 나눕니다.

- 해당하는 각 노드에는 4개의 사분면에 해당하는 4개의 자식 노드가 있습니다.

§ 리프 노드는 0과 고정된 최대 포인트 수 사이에 있습니다(예: 1로 설정).



R-트리

§ R-트리는 B+-트리 의 N차원 확장으로 직사각형 및 기타 다각형 세트를 인덱싱하는 데 유용합니다. § R+
와 같은 변형과 함께 많은 최신 데이터베이스 시

스템에서 지원됨 -
트리 및 R*-트리.

§ 기본 아이디어: 각 B+ 트리 노드와 관련된 1차원 간격의 개념을 N차원 간격, 즉 N차원 사각형
으로 일반화합니다.

§ 2차원 사례(N = 2)만 고려합니다. • R-트리가 잘 작동하지만 N > 2
에 대한 일반화는 간단합니다.
상대적으로 작은 N에 대해서만

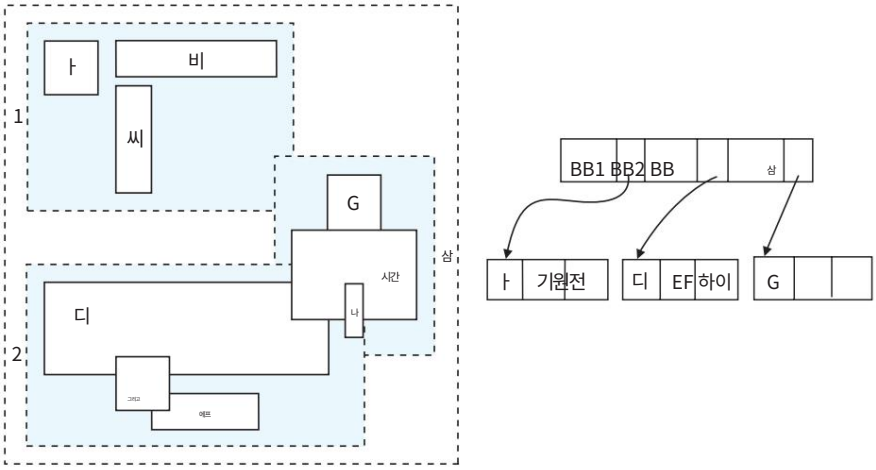
§ 노드의 경계 상자는 다음을 포함하는 최소 크기의 사각형 입니다 .
노드와 관련된 모든 사각형/다각형

• 노드 하위의 경계 상자는 겹칠 수 있습니다.

예 R-트리

§ 직사각형 세트(실선)와 경계 상자(점선)
직사각형에 대한 R-트리의 노드.

§ 오른쪽에 R-트리가 표시됩니다.

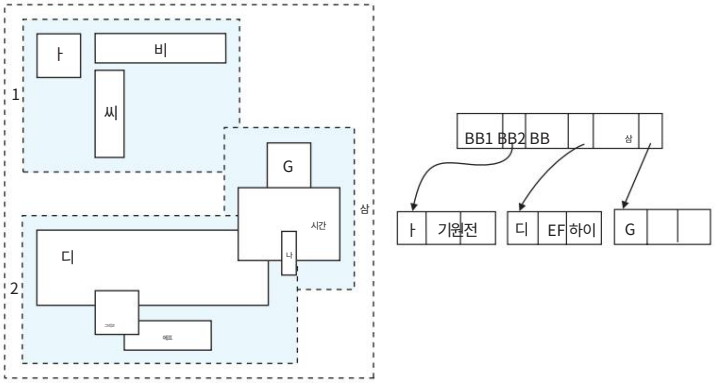


R-트리에서 검색

§ 주어진 쿼리 지점/영역과 교차하는 데이터 항목을 찾으려면 다음을 수행합니다.
루트 노드에서 시작:

- 노드가 리프 노드인 경우 주어진 쿼리 지점/영역과 키가 교차하는 데이터 항목을 출력합니다.
- 그렇지 않으면 경계 상자가 쿼리 포인트/영역과 교차하는 현재 노드의 각 자식에 대해 재귀적으로 자식을 검색합니다.

§ 다중 경로가 필요할 수 있으므로 최악의 경우 매우 비효율적일 수 있습니다.
검색했지만 실제로는 만족스럽게 작동합니다.



임시 데이터 인덱싱

§ 임시 데이터는 관련 기간(간격)이 있는 데이터를 의미합니다.

- 예: 과정 관계의 임시 버전

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

§ 시간 간격에는 시작 및 종료 시간이 있습니다.

- 튜플이 현재 유효하고 유효 종료 시간을 현재 알 수 없는 경우 종료 시간을 무한대로 설정(또는 9999-12-31과 같은 큰 날짜)

§ 쿼리는 특정 시점 또는 일정 기간 동안 유효한 모든 튜플을 요청할 수 있습니다.
시간 간격

- 유효한 기간에 대한 인덱스로 이 작업 속도 향상

임시 데이터 인덱싱(계속) Data (Cont.)

§ 속성 a에 시간 인덱스를 생성하려면: • 속성 a를 하나의
차원으로, 시간을 또 다른 차원으로 하는 R-트리와 같은 공간 인덱스를 사용합니다.

§ 유효한 시간은 시간 차원에서 간격을 형성합니다.

• 현재 유효한 튜플은 값이 무한하기 때문에 문제를 일으킵니다.
또는 매우 큰

§ 솔루션: 모든 현재 튜플(종료 시간을 무한대로)을 (a, 시작 시간)에 색인된 별도의 인덱스에 저장합니다.

• 현재 튜플 인덱스에서 t 시점에 유효한 튜플을 찾으려면 (a, 0)에서 (a,t) 범위의 튜플을 검색합니다.

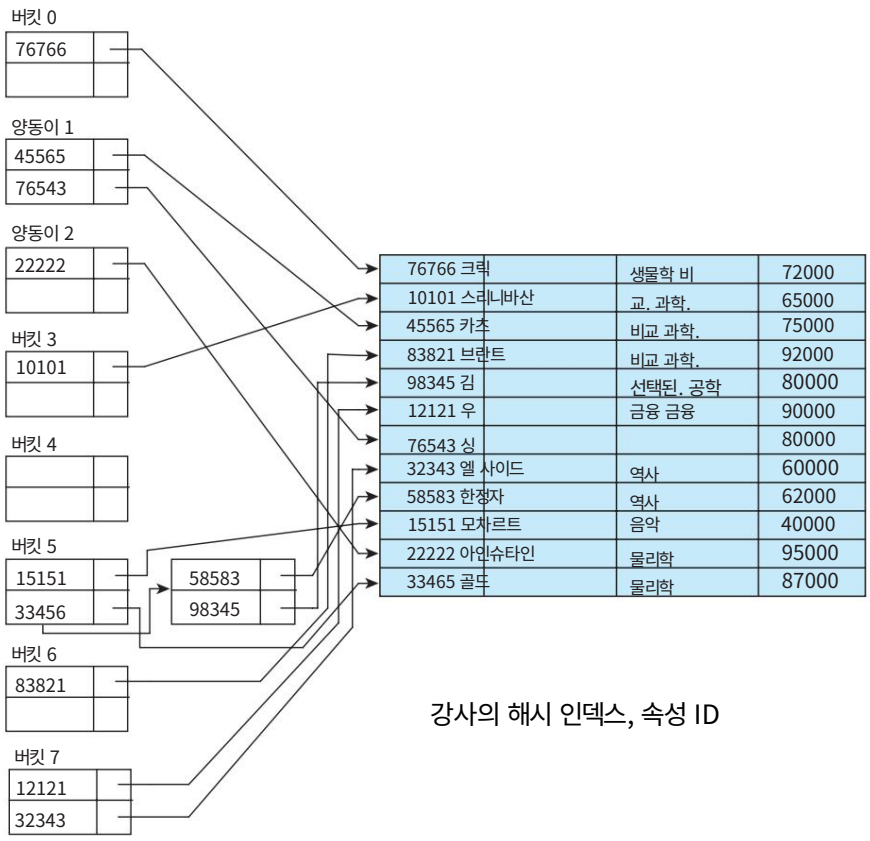
§ 기본 키에 대한 임시 인덱스는 임시 기본 키를 적용하는 데 도움이 될 수 있습니다.
강제

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

14장 끝 Chapter 14

해시 인덱스의 예

Example of Hash Index



강사의 해시 인덱스, 속성 ID