Database

Lecture 5. Indexing

Spring 2024

Prof. Jik-Soo Kim, Ph.D.

E-mail: jiksoo@mju.ac.kr

Notes

- **Readings**
  - Chapter 14: Indexing (Database System Concepts 7^{th} Edition)

BASIC INDEXING MECHANISMS

Indexing Basic Concepts

Clustering/Secondary Indices

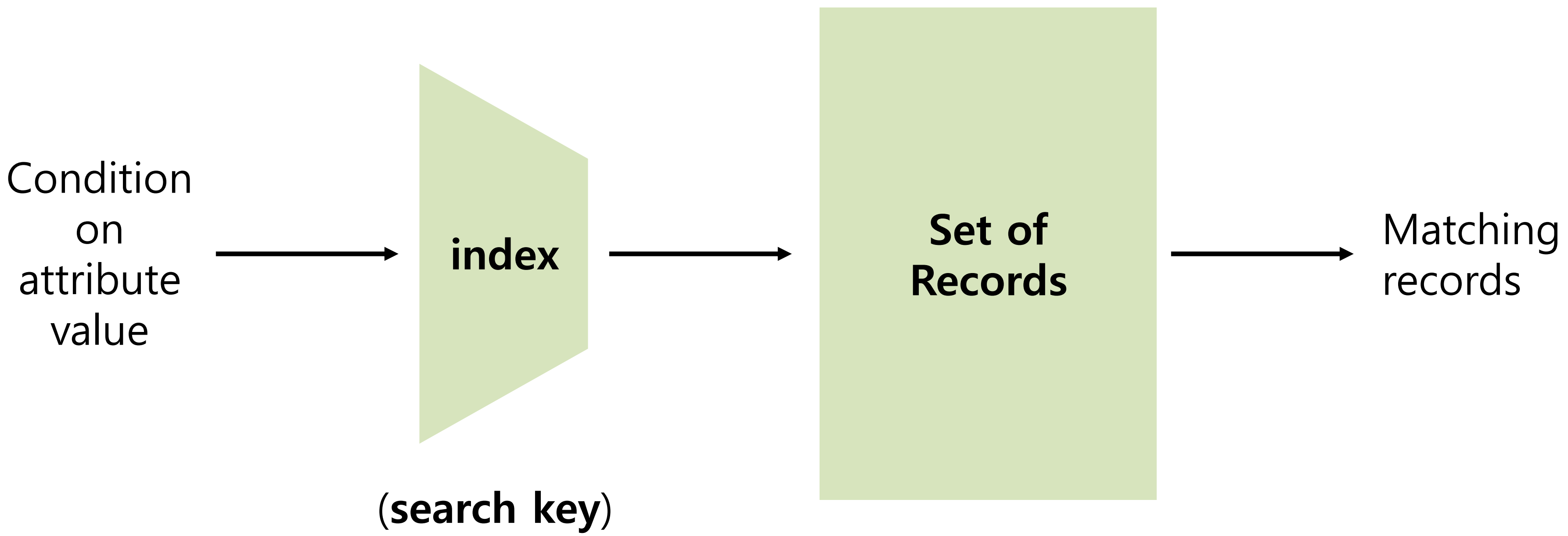
Basic Concepts

- Many queries reference **only a small proportion** of the records in a file!
  - "Find all instructors in the Physics department"
  - "Find the total number of credits earned by the student with ID 22201"

- Ideally, the system should be able to locate these records *directly*
  - to allow these forms of access, we design *additional* structures that we associate with files 다이렉트로 빠르게 찾을 수 있는게 index

Basic Concepts

An **index** is a data structure that supports *efficient* access to data



Basic Concepts

- An index for a file in a database system works in much the same way as the index in this textbook
  ① search for the topic in the index at the back of the book
  ② find the pages where it occurs
  ③ read the pages to find the information for which we are looking

- The words in the index are in **sorted order**, making it easy to find the word we want

- The index is **much smaller** than the book, further reducing the effort needed

Basic Concepts

- Indexing are used to <u>speed up access to desired data</u>
  - e.g., author catalog in library
- **Search Key** - *attribute* or *a set of attributes* used to look up records in a file 프라이머리 키 제외하고 index는 사용자가 만들어야된다.
- An **index file** consists of records (called **index entries**) of the form

search-key (value)	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function"

Index Evaluation Metrics

- **Access types**
  - finding records with a specified attribute value
  - finding records whose attribute values fall in a specified range

특정한 값을 가지거나 범위에 들어가는 값

- **Access time**
  - the time it takes to find a particular data item, or a set of items
- **Insertion time**
  - the time it takes to insert a new data item
    - new data insertion + index structure update
- **Deletion time**
  - the time it takes to delete a data item
    - data deletion + index structure update
- **Space overhead**
  - the additional space occupied by an index structure

Ordered Indices

- In an **ordered index,** index entries are stored *sorted* on the search key value
  - just like the index of a book or a library catalog
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file
  - also called **Clustering index**
  - the search key of a primary index is usually but not necessarily the primary key 대개의 경우 프라이머리키를 가지고 만든다.
- **Secondary index**: an index whose search key specifies an order *different* from the sequential order of the file
  - also called **Non-clustering index** index를 만들었으나 정렬기준이 상관이없다면 secondary index
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key

즉 정렬되서 나오냐 안나오냐가 primary secondary갈림

Dense Index Files

- **Dense index** - Index record appears for *every* search-key value in the file index값이 모든값일 경우
  - e.g., index on *ID* attribute of *instructor* relation
    즉 없던 밸류가 생기면 index에서도 추가해야하는게 dnese index.

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

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	

Dense Index Files

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

일대일 매핑이 아니라고 dense가 아닌 것은 아니다.

Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

Sparse Index Files

- **Sparse Index**: contains index records for only *some* search-key values   search가질 수 있는 것중 일부
  - applicable only if records are sequentially ordered on search-key
- To locate a record with search-key value *K*
  - find index record with largest search-key value < *K*
  - search file sequentially starting at the record to which the index record points

10101	
32343	
76766	

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	

즉 primary index는 dense, sparse다 가능

secondary는 dense만 가능(정렬이 안되어있어 index를 통해 찾기 불가)

Pri vs Sec

Den vs Spa

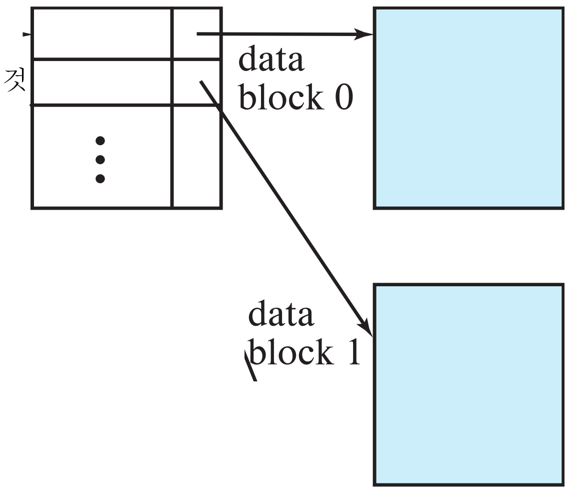
Sparse Index Files

- **Compared to dense indices**

dense보단 느리다.

  - less space and less maintenance overhead for insertions & deletions
  - generally slower than dense index for locating records
- **Good tradeoff**
  - for clustered index: sparse index with an index entry for every *block* in a file, corresponding to least search-key value in the block

어차피 운영체제에서 블럭단위로
가져오기때문에 가장효율적인 방법으로 쓰는 것



  - for unclustered index: sparse index on top of dense index (multilevel index)

Multilevel Index

계층을 쌓듯이 위에 계속 추가하는 것

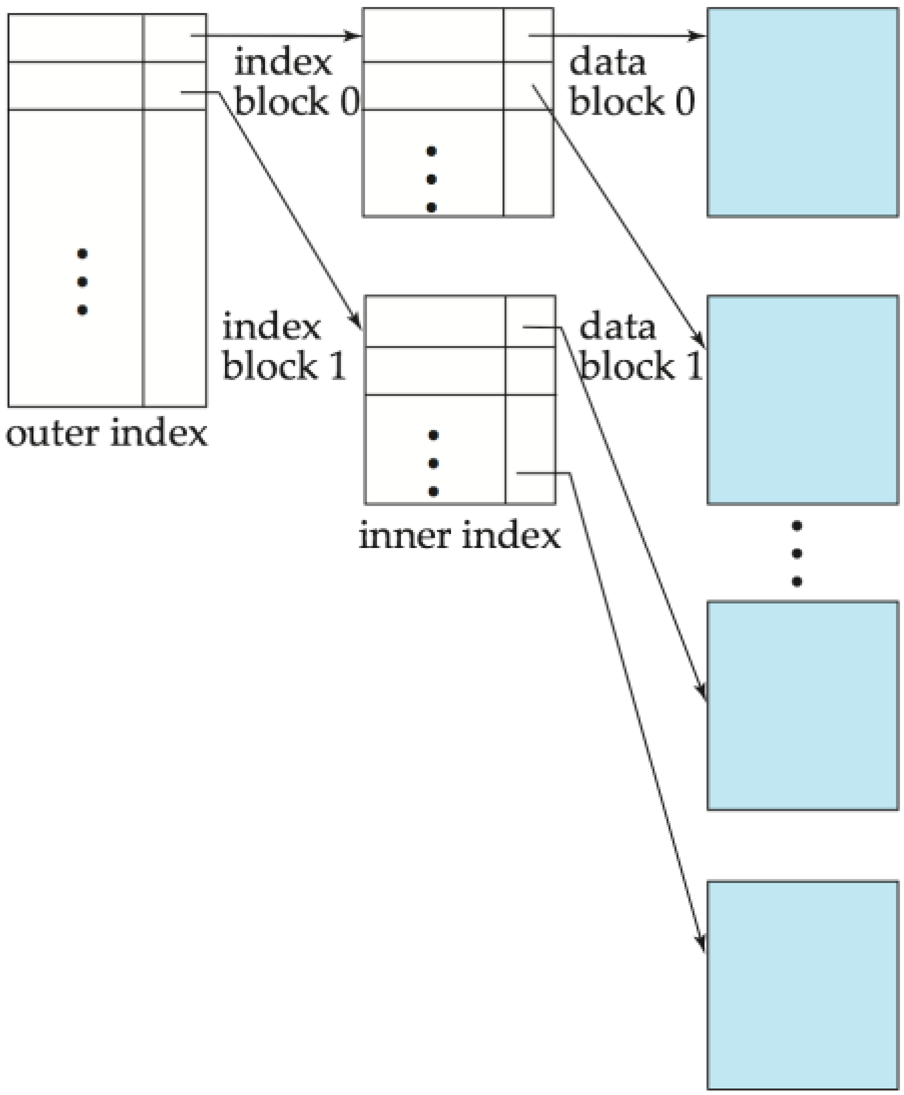
- If index does not fit in memory, access becomes *expensive*

- Solution: treat index kept on disk as a sequential file and construct a sparse index on it
  - outer index: a sparse index of basic index
  - inner index: the basic index file

- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on

- **Indices at all levels must be updated on insertion or deletion from the file!**

Multilevel Index



Index Update: Deletion

- **Dense indices**
  - if the deleted record was the only record with its particular search-key value → delete the corresponding index entry
  - otherwise, update the index entry (modifying the pointers)

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

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	

튜플 중 하나가 지워질경우, 당연히 Dense index도 지워야한다.
반면 안지워도되는게 있는데 중복이 있는 dense(P.K가 아닌 스키마?)는 가능

Index Update: Deletion

- **Sparse indices**
  - if the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done
    - e.g., delete the Wu or Singh
  - otherwise, update the index entry to point to the next record

10101	
32343	
76766	

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	

ID가 13000 vs 9000
13000이 들어오면 최솟값이랑상관이없어
변화X
하지만 9000이면 블럭의 최솟값이므로
index수정필요하다.

Index Update: Insertion

- **Single-level index insertion**
  - perform a lookup using the search-key value appearing in the record to be inserted
  - **Dense indices**
    1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position

    2. Otherwise, the following actions are taken
       a. If the index entry stores pointers to all records with the same search-key value, the system adds a pointer to the new record in the index entry
       b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values

즉 새로운 튜플이 새로 들어와도, 어떻게 정렬되어있냐에 따라 매커니즘이 달라질 수 있다.

Index Update: Insertion

- **Single-level index insertion**
  - **Sparse indices**: assume that the index stores an entry for each block
    1. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index

    2. On the other hand,
       a. if the new record has the least search-key value in its block, the system updates the index entry pointing to the block
       b. if not, the system makes no change to the index

- **Multilevel insertion and deletion** algorithms are simple extensions of the single-level algorithms

여기까지 나온 index모두 primary index였다.

Secondary Indices

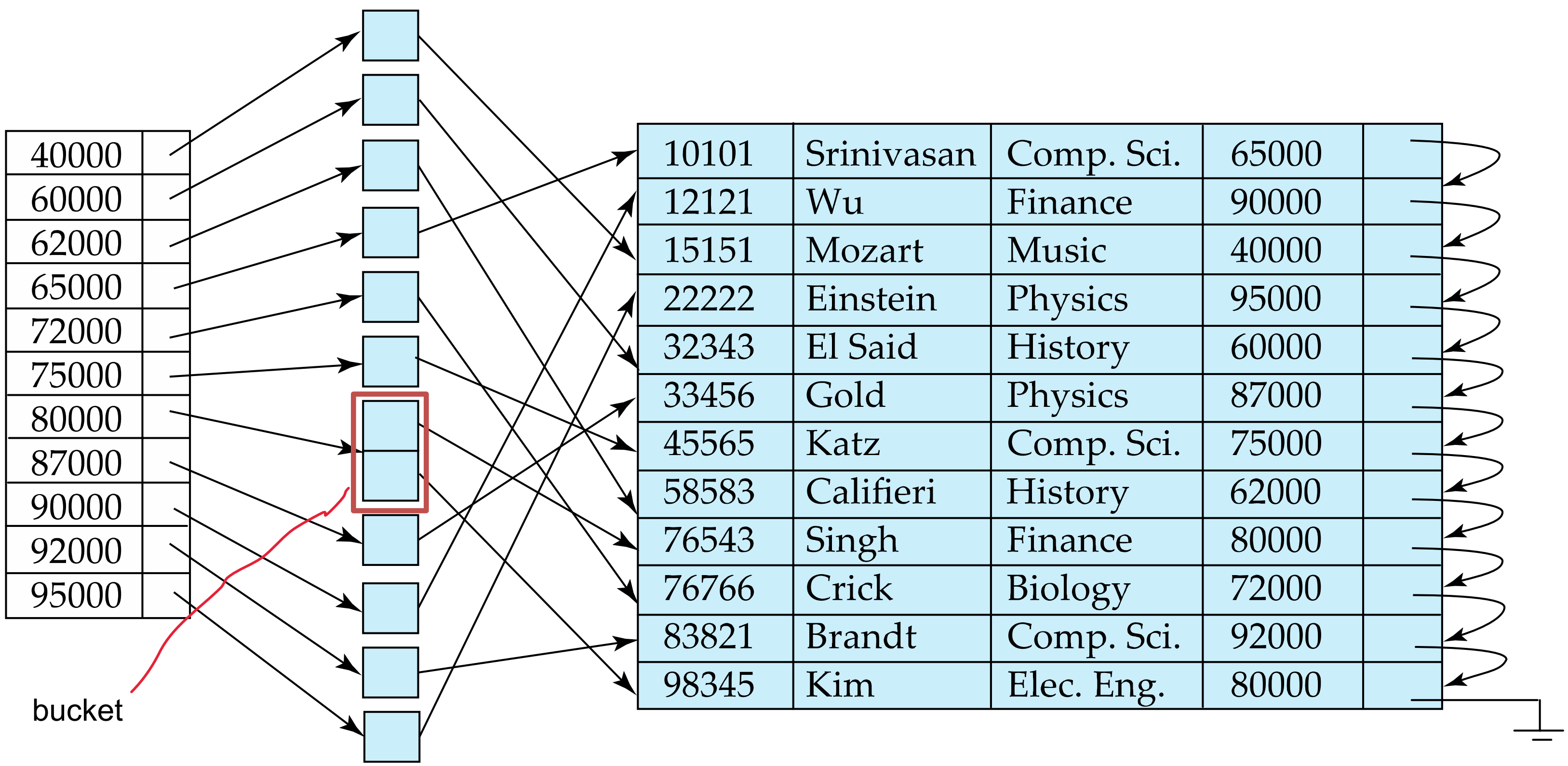
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index)
  - Example 1: in the *instructor* relation stored sequentially by *ID*, we may want to find all instructors in a particular *department*
  - Example 2: as above, but where we want to find all *instructors* with a specified *salary* or with *salary* in a specified range of values

- We can have a **secondary index** with an index record for each search-key value

secondary index는 무조건 dense여야 한다.

Secondary Indices Example

- Index record points to a "bucket" that contains *pointers* to all the actual records with that particular search-key value
- Secondary indices have to be **dense**



Secondary index on *salary* field of *instructor*

Primary and Secondary Indices

- Indices offer substantial benefits when *searching* for records
- BUT: Updating indices imposes *overhead* on database modification → <u>when a file is modified, every index on the file must be updated!</u>
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive on magnetic disk
  - each record access may fetch a new block from disk
  - each block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

Automatic Creation of Indices

- Most database implementations *automatically* create an index on the **primary key**
  - whenever a tuple is inserted into the relation, the index can be used to <u>check that the primary key constraint is not violated</u>
  - without the index on the primary key, whenever a tuple is inserted, the *entire* relation would have to be read!

B+-TREE INDEX FILES

B⁺-Tree Index Files

B$^+$-tree indices are an alternative to index-sequential files

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, both for index lookups and sequential scans since many *overflow* blocks get created
  - periodic *reorganization* of entire file is required
- Advantage of B$^+$-tree index files
  - automatically reorganizes itself with small, local changes in the face of insertions and deletions 자동적으로 작은 수정사항도 반영 즉, 전체를 바꿀일이 없다.
  - reorganization of entire file is not required to maintain performance
- (Minor) disadvantage of B$^+$-trees
  - extra insertion and deletion overhead, space overhead

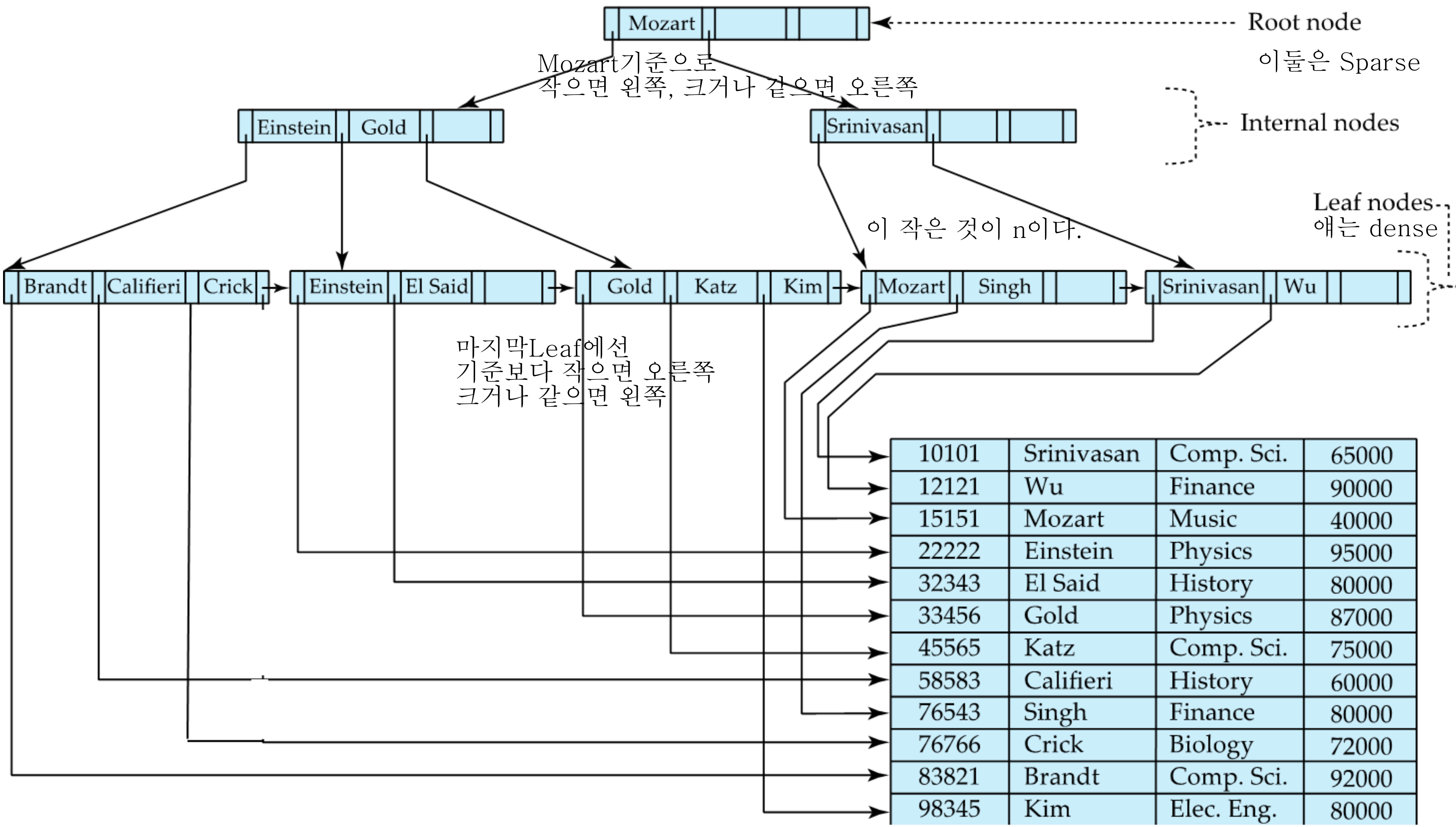
- **Advantages of B$^+$-trees outweigh disadvantages!**
  - B$^+$-trees are used extensively

    널리 사용된다.

Example of B^+-Tree

name이라는 어트리뷰트를 search-key로 지정해 indexing함.

이건 secondary table이ek. (P.k는 교번인데 name사용)



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

B⁺-Tree Index Files

B⁺-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the *same* length 트리구조가 대칭적
- 중간 노드들(internal)
  Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children (n = number of pointers in a node) 포인터개수
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
  최소 n−1 /2 만큼은 채워져야한다.

인덱싱을 효과적으로 하기위한 조건이라 보면됨.

- 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⁺-Tree Node Structure

- Typical node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  – K_i are the search-key values
  – P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)

동일 밸류들이 있더라도 똑같이 노드에 동일 밸류들을 작성한다.(기영이가 3명일경우 3개노드그대로씀)

- The search-keys in a node are *ordered*

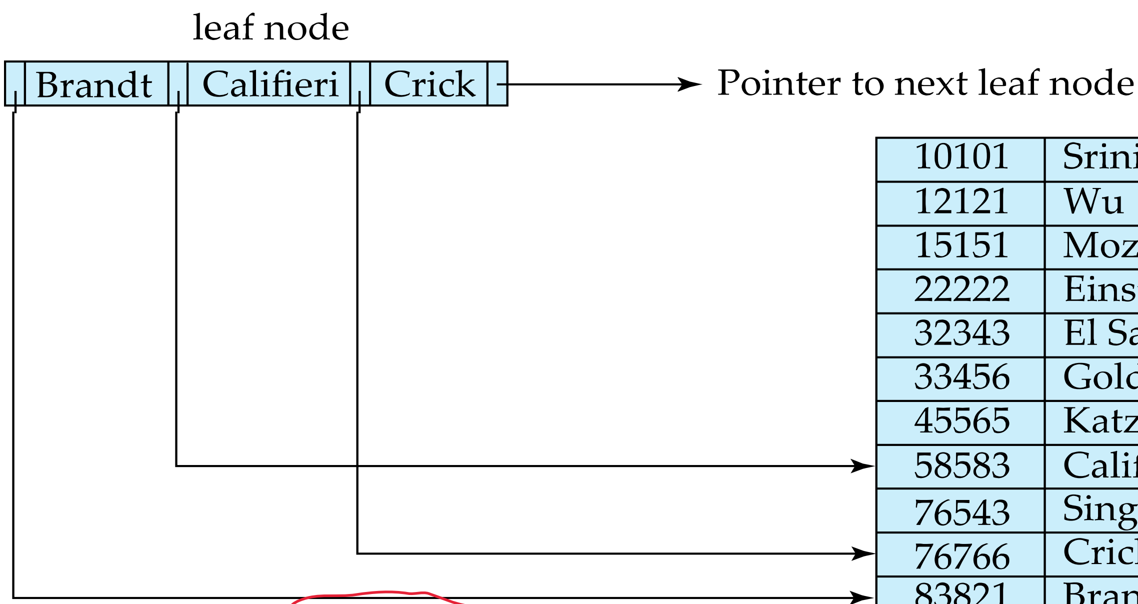
$$K_1 < K_2 < K_3 < . . . < K_{n–1}$$

(Initially, assume no duplicate keys, address duplicates later)

Leaf Nodes in B$^+$-Trees

Properties of a leaf node:

- For i = 1, 2, . . ., n–1, pointer P_i points to a file record with search-key value K_i
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



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

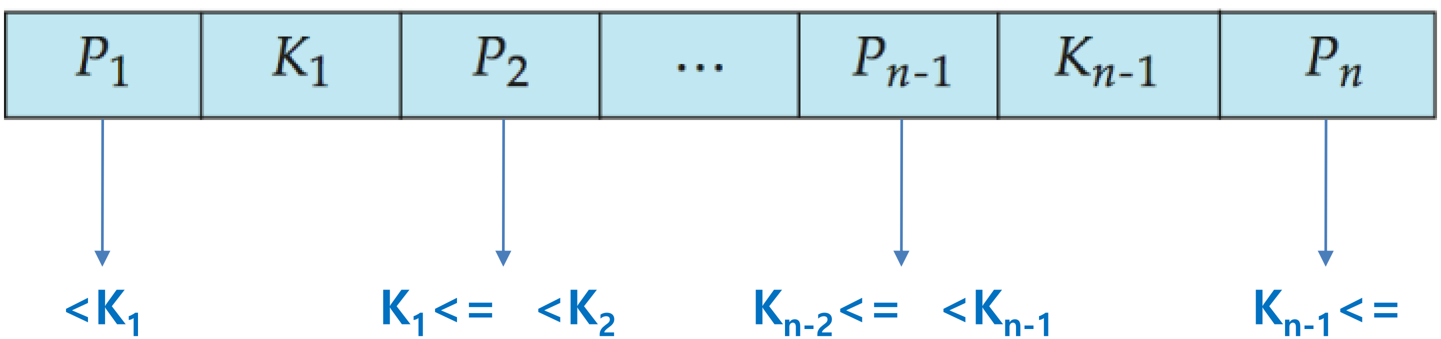
***dense* index**

MYONGJI UNIVERSITY

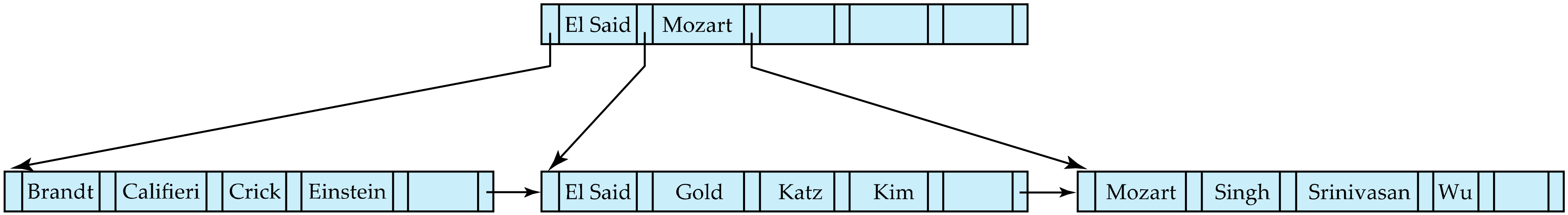
Non-Leaf Nodes in B^+-Trees

분류기준

- **Non leaf nodes** form a **multi-level *sparse* index** on the leaf nodes. For a non-leaf node with *n* pointers:
  - all the search-keys in the sub-tree to which P_1 points are less than K_1
  - for 2 <= i <=n – 1, all the search-keys in the sub-tree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - all the search-keys in the sub-tree to which P_n points have values greater than or equal to K_{n-1}



Example of B⁺-tree

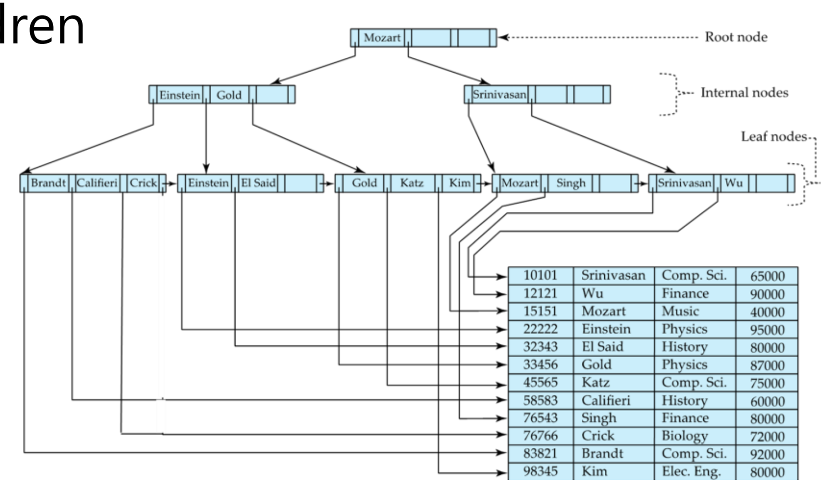


B⁺-tree for *instructor* file (***n*** **= 6**)

- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with n = 6)
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2) \rceil$ and n with n =6)
- Root must have at least 2 children

n이 클수록 성능 up!!

vs. when *n* = 4



Observations about B⁺-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close!
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices
- The B^+-tree contains a relatively *small* number of levels
  - level below root has at least $2 * \lceil n/2 \rceil$ values
  - next level has at least $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
- If there are *K* search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ thus searches can be conducted efficiently
  - insertions and deletions to the main file can be handled efficiently, as the index can be restructured in *logarithmic* time

    즉 search key가 아무리 많아도 log함수형태이므로 시간은 오래안걸린다.

기말 낼것! (틀린부분찾아 바꾸시오로)

Queries on B$^+$-Trees

function *find*(*v*) ex. V= mozart, wu, brandt일때

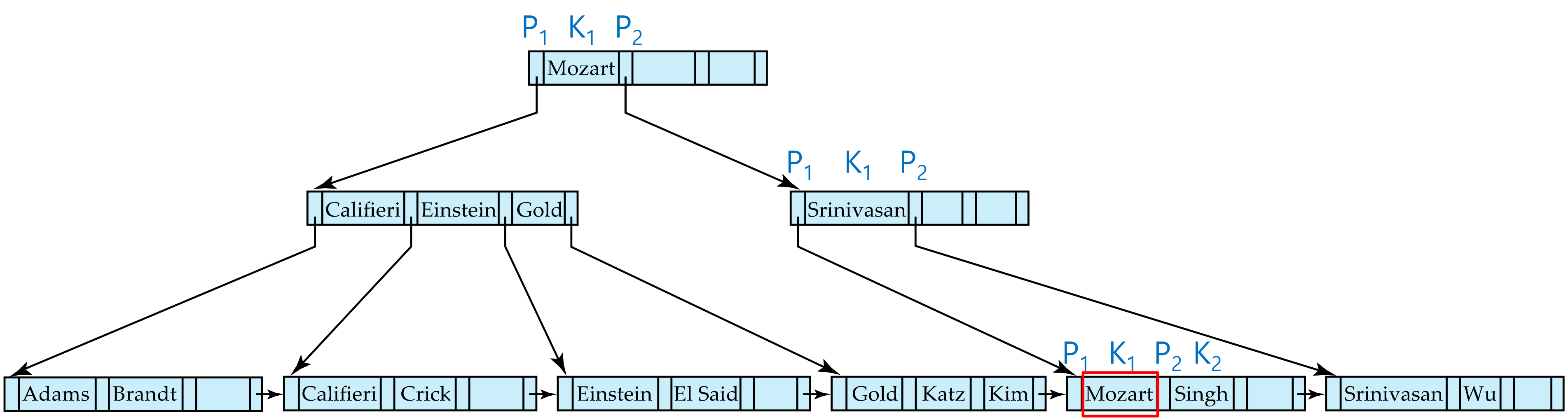
Typical B+-tree node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

1. *C = root*
2. **while** (C is not a leaf node)
   1. Let *i* be least number s.t. $V \leq K_i$.
   2. **if** there is no such number *i* then
   3. Set *C = last non-null pointer in C*
   4. **else if** (v = C.K_i) Set C = P_{i+1}
   5. **else set** *C* = C.P_i
3. **if** for some *i*, $K_i = V$ **then** return C.P_i
4. **else** return null /* no record with search-key value *v* exists. */ 없다면 null반환

주어진 키를따라 작으면 왼쪽
크거나 같으면 오른쪽!

마지막 바닥에 왔을때만 같으면 왼쪽!



Queries on B+-Trees

```
function find(value V)
/* Returns leaf node C and index i such that C.Pi points to first record
* with search key value V */
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that V ≤ C.Ki
        if there is no such number i then begin
            Let Pm = last non-null pointer in the node
            Set C = C.Pm
        end
        else if (V = C.Ki)
            then Set C = C.Pi+1
        else C = C.Pi /* V < C.Ki */
    end
    /* C is a leaf node */
    Let i be the least value such that Ki = V
    if there is such a value i
        then return (C, i)
        else return null ; /* No record with key value V exists*/
```

같은 search key value V를
갖는 레코드들이 복수개
있을 수 있음!

Queries on B+-Trees

Handling Duplicates: fetch all records with a specified search key *V*

procedure printAll(*value* V)
/* prints all records with search key value V */
  Set done = false;
  Set (L, i) = find(V);
  if $((L, i)$ is null) **return**
  repeat
    repeat
      Print record pointed to by $L.P_i$
      Set $i = i + 1$
    until ($i >$ number of keys in L **or** $L.K_i > V$)
    if ($i >$ number of keys in L)
      then $L = L.P_n$
      else Set done = true;
  until (done **or** L is null)

노드 *L* 내에서 Traverse

다음 노드도 검색 필요

테이블 넘어가는 것

모든 *V* 값 레코드 검색 완료

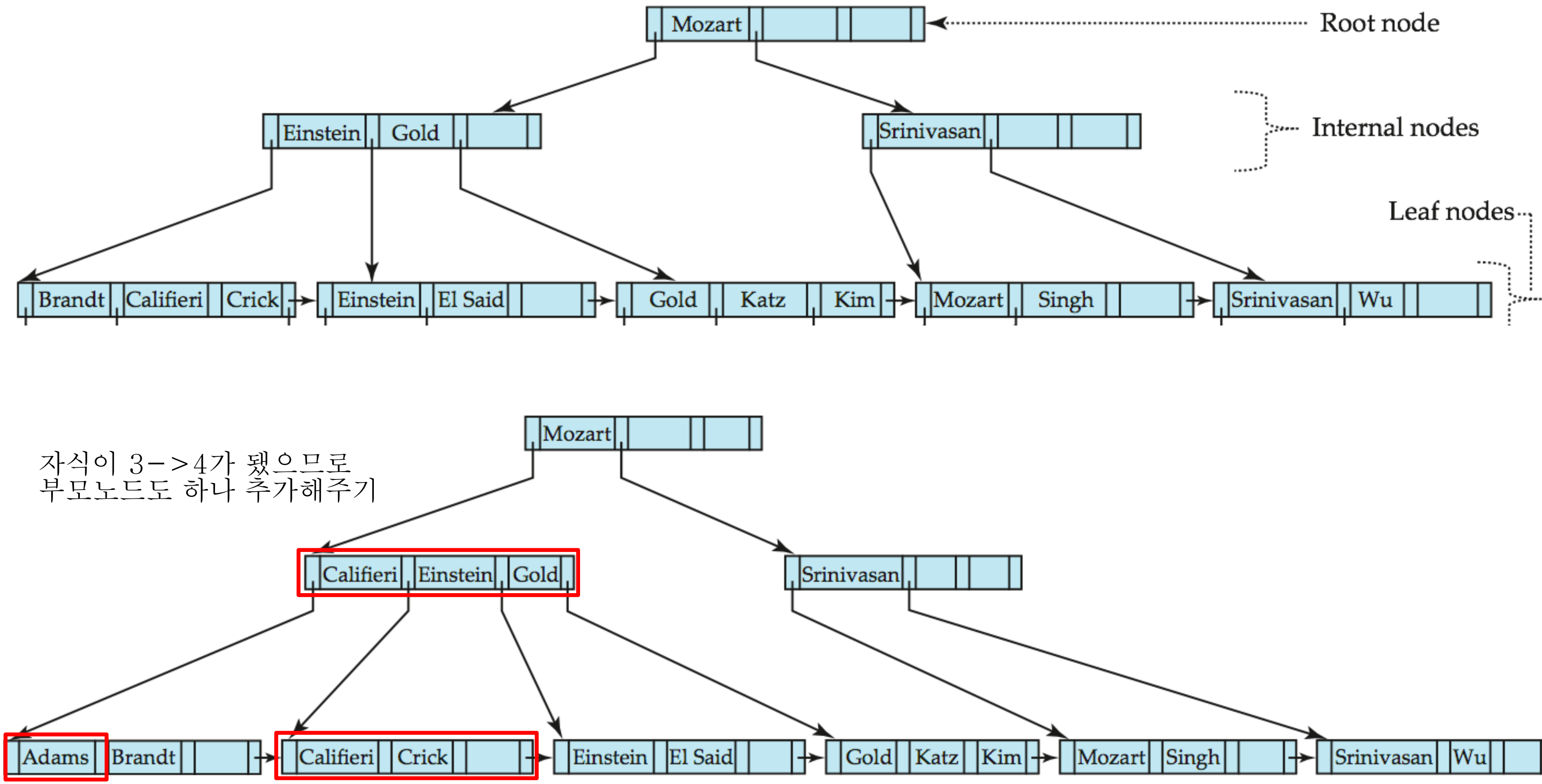
예) | Mozart | Mozart | Mozart | → | Mozart | Mozart | Singh |

Queries on B^+-Trees

- If there are *K* search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ 이때, n은 # of children(자식의 개수)
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and *n* is typically around 100 (40 bytes per index entry)
- With 1 million search key values and *n* = 100
  - at most $\log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds!

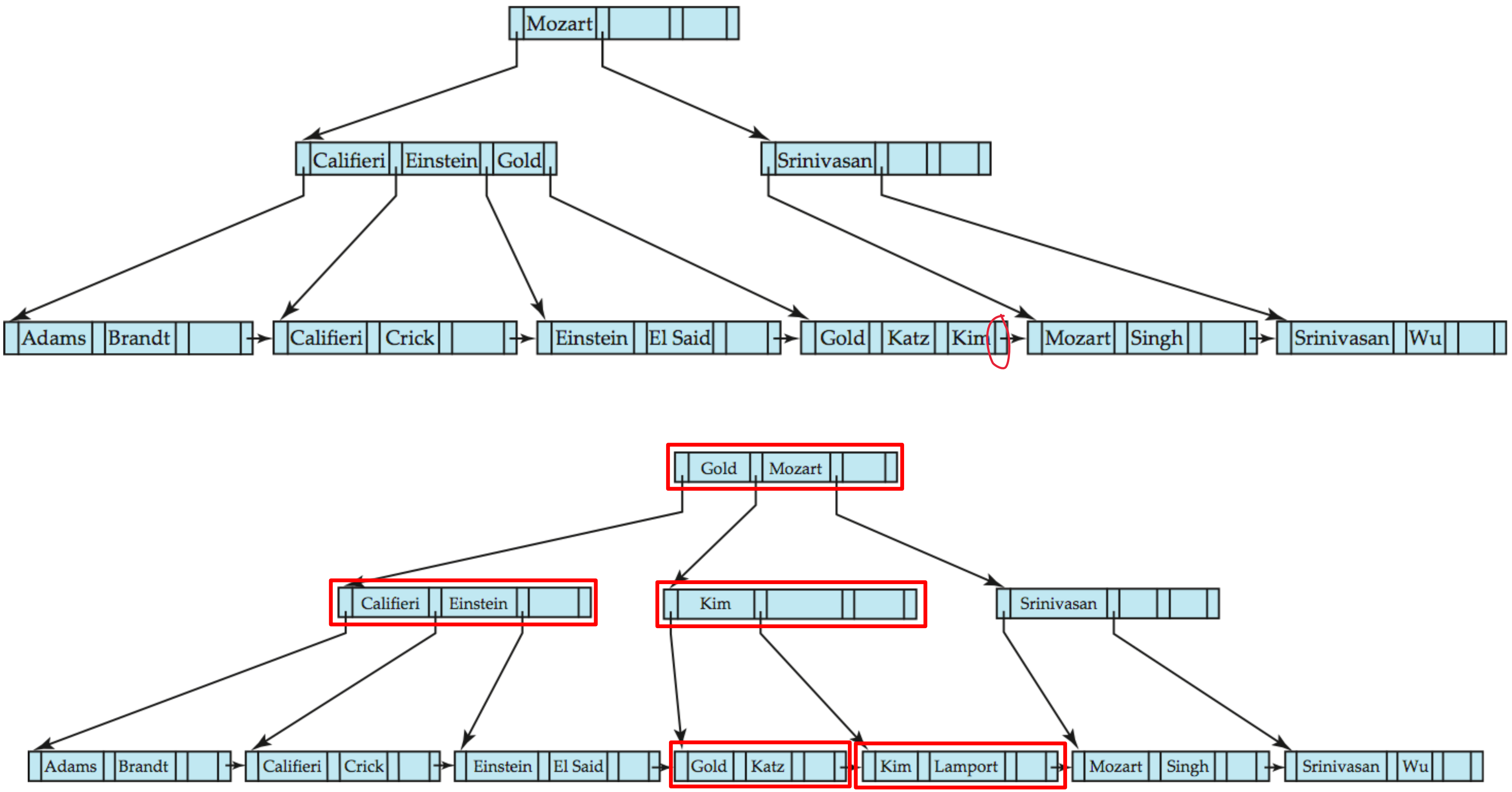
Updates on B+-Trees: Insertion

느낌만 알기 시험X



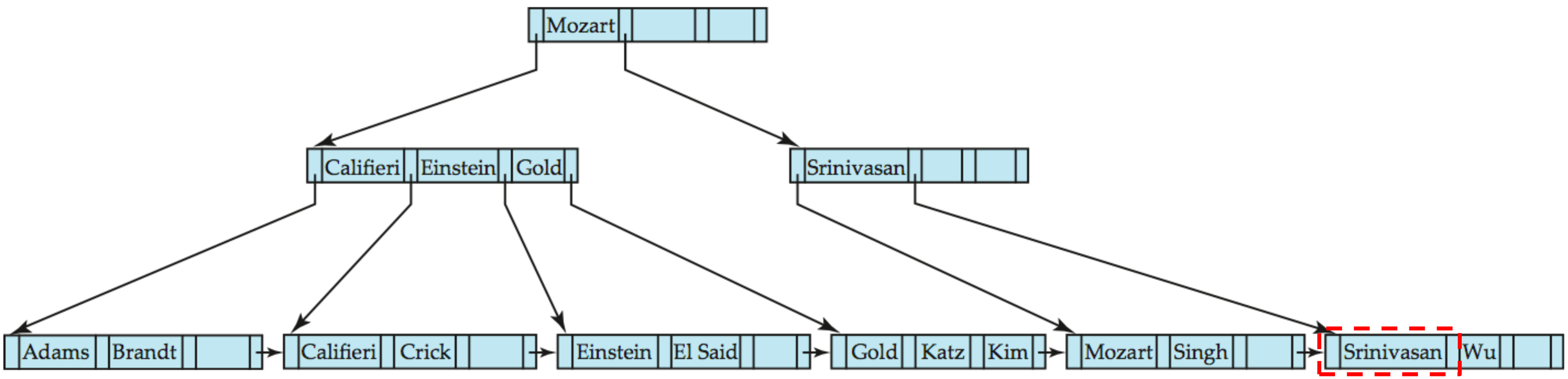
B+-Tree before and after insertion of “Adams”

Updates on B+-Trees: Insertion

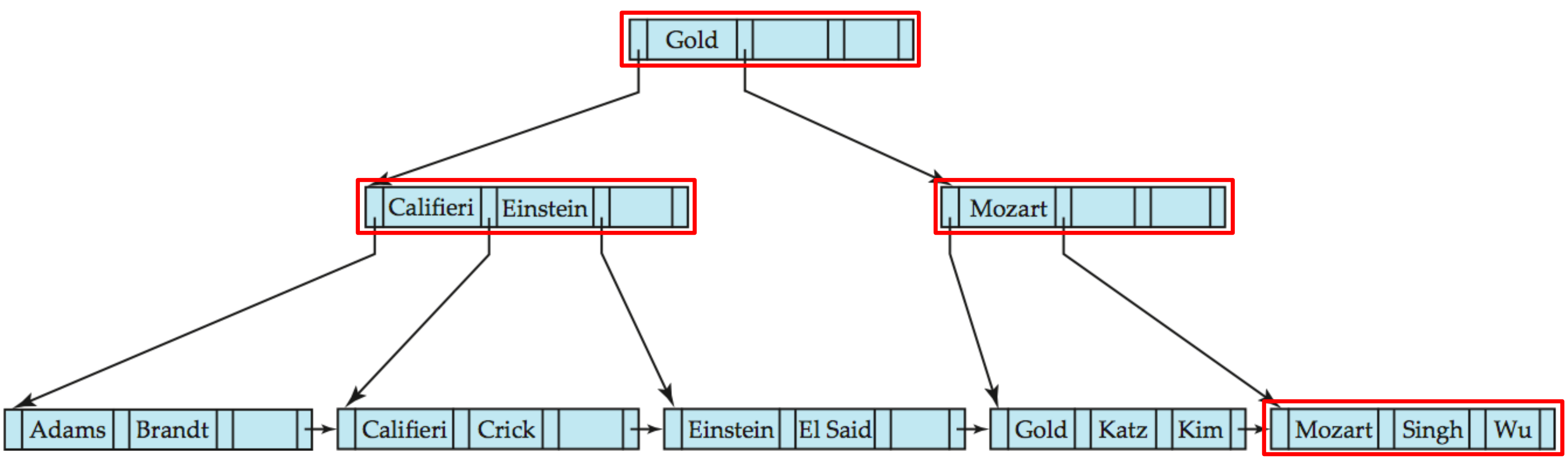


B^{+}-Tree before and after insertion of "Lamport"

Updates on B+-Trees: Deletion

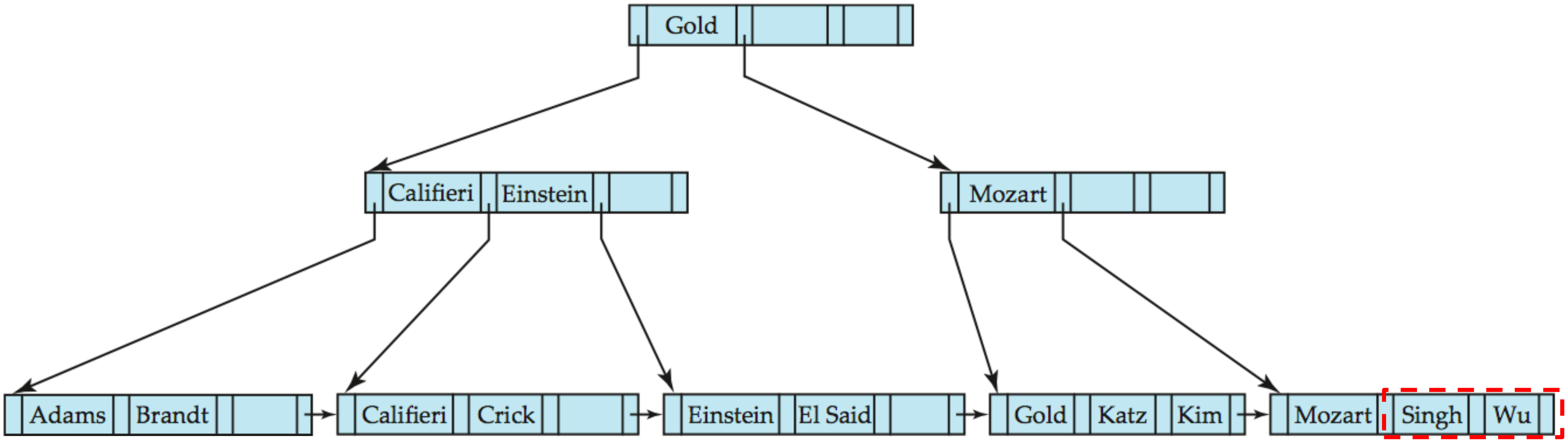


Before and after deleting "Srinivasan"

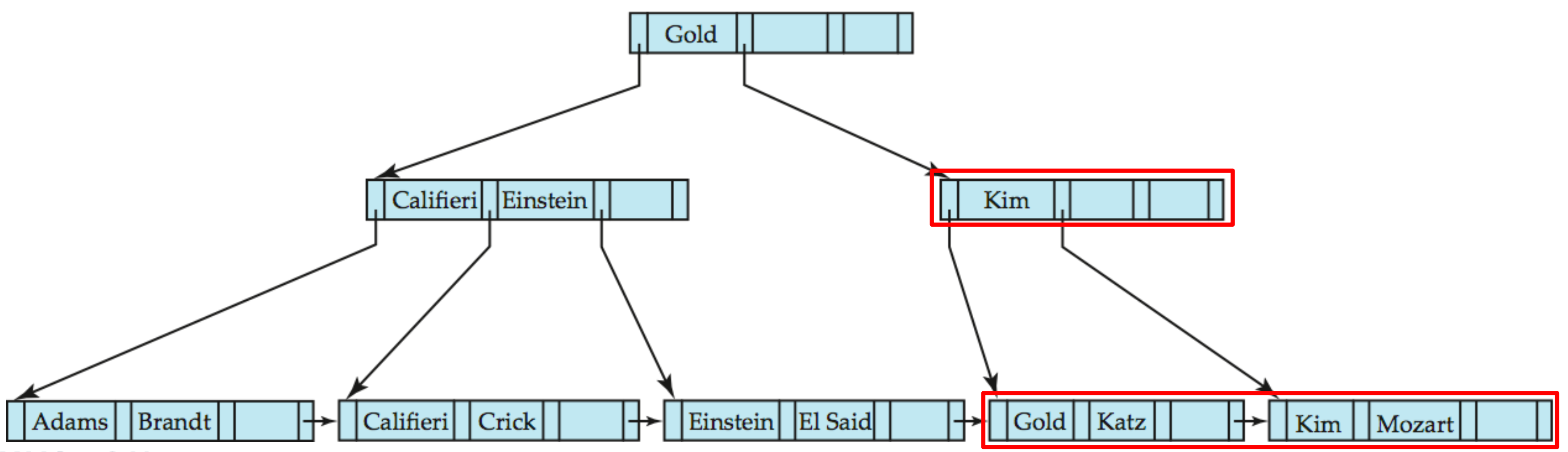


Deleting "Srinivasan" causes *merging* of under-full leaves

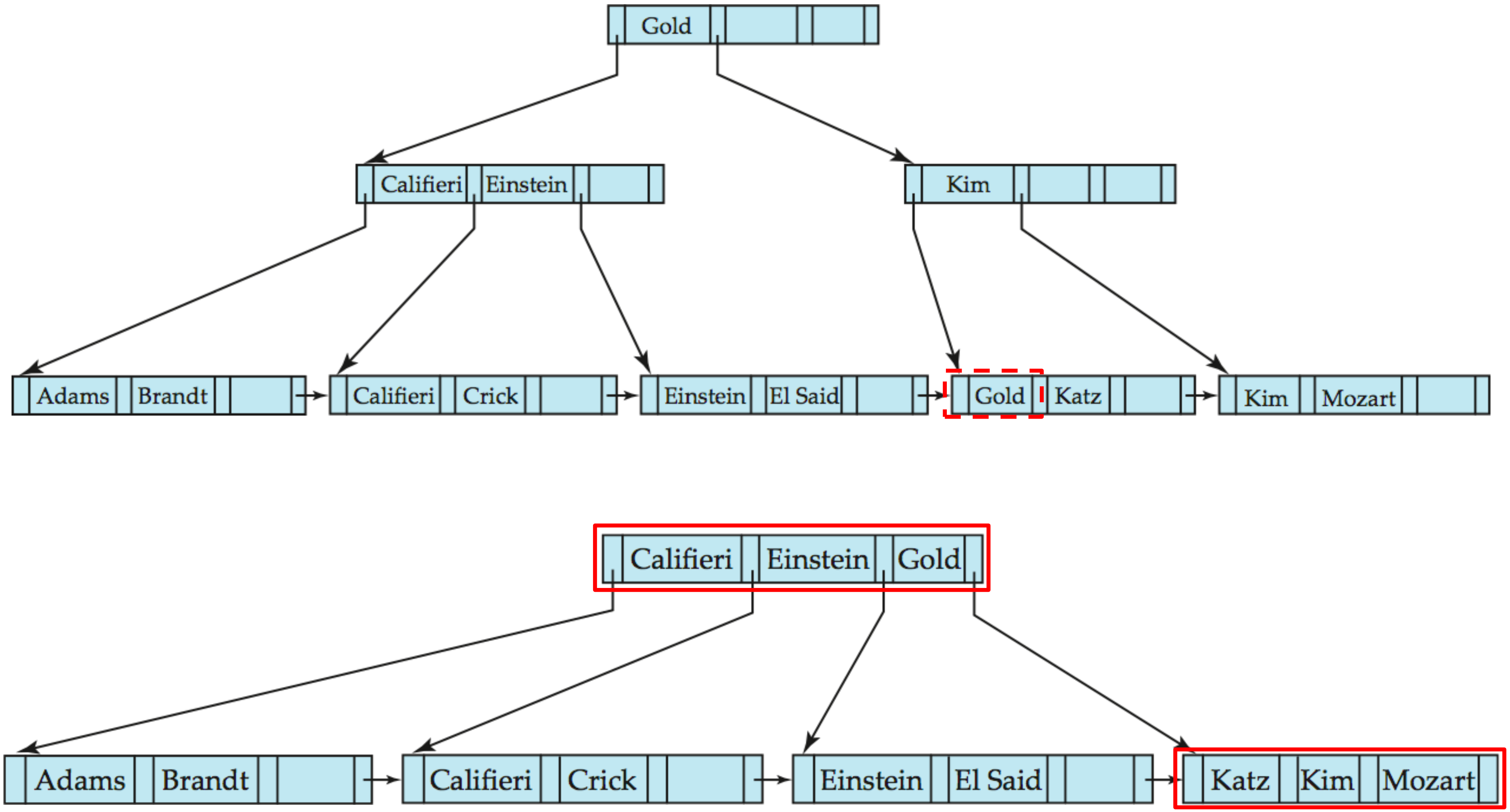
Updates on B+-Trees: Deletion



Deletion of “Singh” and “Wu”



Updates on B+-Trees: Deletion



Before and after deletion of “Gold”

B^+-TREE EXTENSIONS

B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear *only once*; eliminates redundant storage of search keys
- Search keys in non-leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a non-leaf node must be included

B+ tree > Bitmap

Generalized B-tree leaf node

P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

(a)

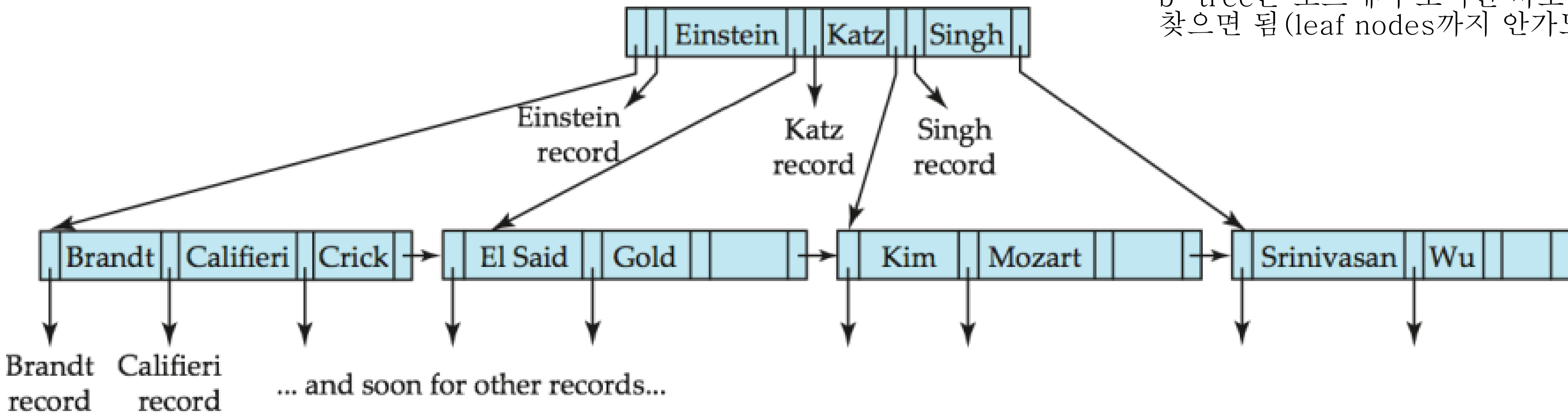
P_1	B_1	K_1	P_2	B_2	K_2	$\cdots$	P_{m-1}	B_{m-1}	K_{m-1}	P_m

(b)

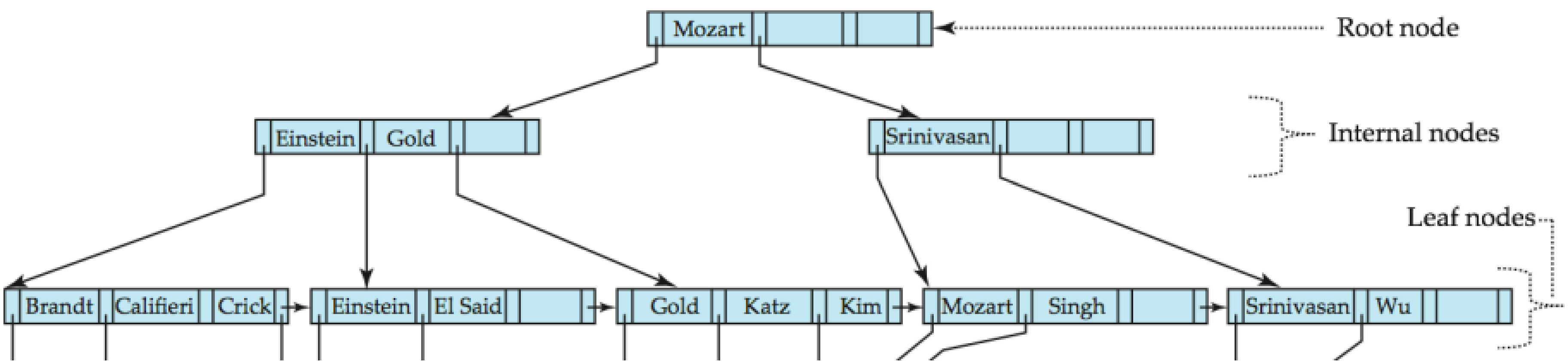
Non-leaf node – pointers B_i are the bucket or file record pointers

B-Tree Index File Example

b−tree는 노드에서 보이면 바로
찾으면 됨(leaf nodes까지 안가도 됨)



B-tree (above) and B^{+}-tree (below) on same data



B-Tree Index Files

- **Advantages of B-Tree indices:**
  - may use less tree nodes than a corresponding B^+-Tree 당연 노드수가 적다.
  - sometimes possible to find search-key value before reaching leaf node
- **Disadvantages of B-Tree indices:**
  - only a small fraction of all search-key values are found early
  - 자식이 줄어든다.
  - non-leaf nodes are larger, so fan-out is reduced → B-Trees typically have greater depth than corresponding B^+-Tree
  - insertion and deletion are more complicated than B^+-Trees
  - implementation is harder than B^+-Trees

  크다는 것은 정보가 많다고 보면 됨.
- **Typically, advantages of B-Trees do not outweigh disadvantages!**

MULTI-KEY ACCESS & BITMAP INDICES

Multiple-Key Access

- Use *multiple* indices for certain types of queries
- Example: two indices (*dept_name*, *salary*)

  select *ID*

  from *instructor*

  where *dept_name* = "Finance" **and** *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
  1. Use index on *dept_name* to find instructors with department name "Finance"; test *salary* = 80000
  2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = "Finance".
  3. Use *dept_name* index to find pointers to all records pertaining to the "Finance" department; Similarly use index on *salary*, Take intersection of both sets of pointers obtained.

Multiple-Key Access

결과는 가져오나 느리게 가져온다는 것

- All three strategies to process the query may not work well if …
  - there are many records pertaining to the Finance department
  - there are many records pertaining to instructors with a salary of \$80,000
  - there are only a few records pertaining to both the Finance department and instructors with a salary of \$80,000

  즉 index를 사용하는 의의를 생각해야 댐.

- An index structure called **bitmap index** can in some cases speed up the queries involving intersection operation

Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute index를 하나의 스키마가 아닌 두개로 합친 것ㅅ
  - e.g., (*dept_name, salary*)
  - lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
    - $a_1 < b_1$, or 즉, a1인 dept_name먼저깔고 그 뒤 a2인 salary 순서대로
    - $a_1 = b_1$ and $a_2 < b_2$

- Create an index on a composite (concatenated) search key
  - the search key consists of the *department name* concatenated with the *instructor salary*

Indices on Multiple Keys

Suppose we have an index on combined search-key (*dept_name, salary*)

- **where** *dept_name* = "Finance" **and** *salary* = 80000
→ the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions

- Can also efficiently handle
  where *dept_name* = "Finance" **and** *salary* < 80000

- But cannot efficiently handle finance앞 index를 다봐야하므로 sql문과 다를 바 없다. 효율 X
  where *dept_name* < "Finance" **and** *salary* < 80000
  – may fetch many records that satisfy the first but not the second condition

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name> (<attribute-list>)

  e.g., **create index** *b-index* **on** *branch(branch_name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key
  - not really required if SQL **unique** integrity constraint is supported

- To drop an index 지우기

  drop index <index-name>

- Most database systems allow specification of type of index, and clustering

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on *multiple* keys

- A **bitmap index** on the attribute *A* of relation *r* consists of one bitmap for each value that *A* can take

결합 index불가

- Applicable on attributes that take on a relatively small number of *distinct* values
  - e.g., gender, country, state, …
  - e.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)

- A bitmap is simply an array of bits

Bitmap Indices

of bits = # of tuples

- In its simplest form, a bitmap index on an attribute has a bitmap for *each value* of the attribute
  - bitmap has as many bits as records
  - in a bitmap for value *v*, the bit for a record is 1 if the record has the value *v* for the attribute, and is 0 otherwise
  - records in a relation are assumed to be numbered sequentially from, say, 0 → given a number *n* it must be easy to retrieve record *n*

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

0번째 record num이 m이므로
1, 가지지 않으면 0

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - **Intersection** (and)
  - **Union** (or)
  - **Complementation** (not)

논리연산!

- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - e.g., 100110 AND 110011 = 100010
    100110 OR 110011 = 110111
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - can then retrieve required tuples
    - counting number of matching tuples is even faster

Bitmap Indices

- Bitmap indices are generally very small compared to relation size
  - e.g., if a record is 100 bytes, space for a single bitmap is 1/800 of space used by relation
    → if the number of distinct attribute values is 8, bitmap is only 1% of relation size

Efficient Implementation of Bitmap Operations

- Using **bit-wise** instructions
  - a *word* usually consists of 32 or 64 bits
  - a single bit-wise **and** instruction can compute the intersection of 32 or 64 bits at once

  - e.g., if a relation had 1 million records, each bitmap would contain 1 million bits (128 kilobytes)
    - only 31,250 instructions are needed to compute the intersection of two bitmaps, assuming a 32-bit word 32씩 끊어서 31250번을 해야 백만비트연산가능

    → an extremely fast operation!

  - similarly, we can use bit-wise **or**, **not** for bitmap union and **complement** operations 정말 빠른연산!!

Efficient Implementation of Bitmap Operations

- Counting the number of 1s can be done fast by a trick:
  - a precomputed array of 256 elements 2^8
    - the i^{th} entry stores the number of bits that are 1 in the binary representation of i 1의 개수
    - e.g., P[0] = 0, P[1] = 1, P[2] = 1, P[3] = 2, P[4] = 1, P[5] = 2, …

      0000 0001 0010 0011 0100 0101
  - use each byte of the bitmap to index into the precomputed array and add the stored count
    - e.g., 00000001 00000011 00000101 00000100 …

      이 8bits를 정수변환시 p[1]=1 p[3] = 2 p[5]= 2 p[4]= 1....
  - can use pairs of bytes to speed up further at a higher memory cost (e.g., a large array using 2^{16} = 65,536 entries)

MYONGJI UNIVERSITY

Bitmaps and B$^+$-Trees

- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B$^+$-trees
  - for values that have a *large* number of matching records
  - worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits

    x *64bits > T bits
    즉 $x > 1/64 * T$

    B+ tree > Bitmap
  - above technique merges benefits of bitmap and B$^+$-tree indices

Bitmap Index – final points

- Final points about bitmap indices
  - **Relatively low cardinality** makes bitmap indices useful
    - e.g., for bitmap indexes to be built on Zip Code data
    - in this case, the cardinality is absolutely high, but relatively low compared with the millions of possible households

      집주소랑 우편번호 비교시 즉 절대적으론 많지만 상대적으론 작으므로 쓰기 가능.
  - Placing a single bitmap index on a table is pretty pointless?
    - it may still be good
    - bitmap indices derive their usefulness from being combined with other bitmap indices, since it's only when we start performing boolean ANDs and ORs that they start producing results for us

MYONGJI UNIVERSITY

Bitmap Index – final points

- Final points about bitmap indices
  - **Much faster** than B+-Tree indexes **in read-only environments**
  - Bitmap indexes should not be used to create composite column indexes 개별적으로 만들어야한다!
  - in summary, bitmap indices are most appropriate for **large, low-cardinality, read-only, static data tables**!

ORACLE INDEX CASE STUDY

Oracle INDEX

- 종류
  - B^+-Tree: 일반적 인덱스 (트리 기반)
  - Bitmap 인덱스: 특수 용도 (OLAP 등에 사용)

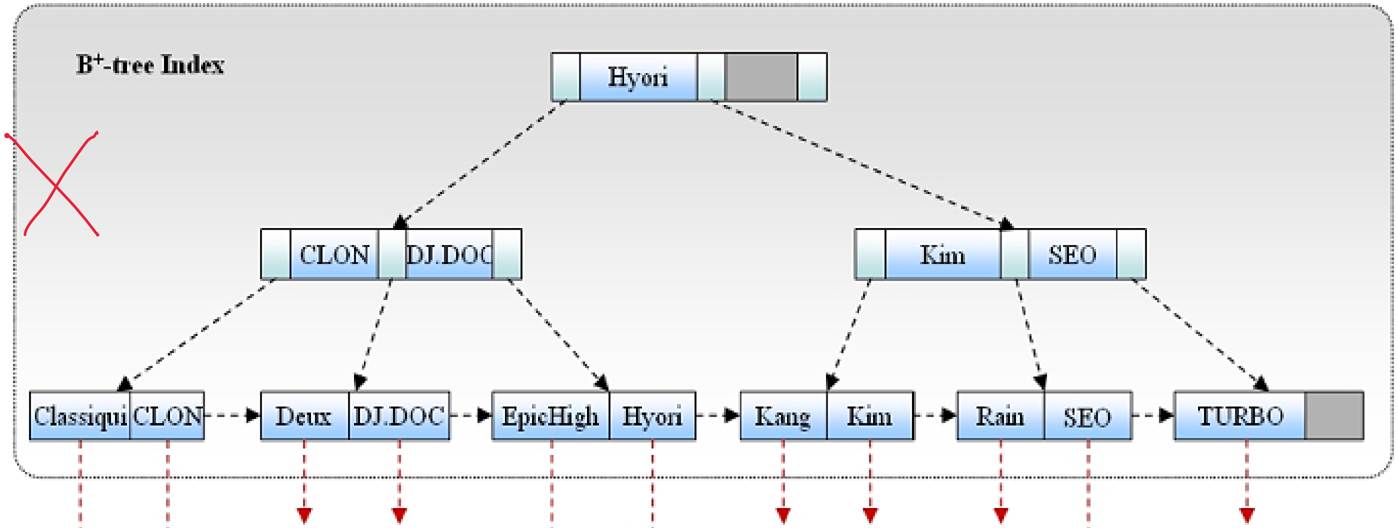
- 구분
  - Single 컬럼 vs. Composite 컬럼
  - Unique vs. Non-Unique
  - Column Data vs. Function-Based
  - Automatic-Created vs. User-Created

B^+-Tree

- 특징
  - 대표적인 트리 기반의 인덱스 구조
  - m-원 탐색 트리
  - Block 기반의 접근 구조 (디스크 저장 구조로 적합)
  - 데이터의 삽입/삭제에도 자동적으로 트리 구조 유지
  - 트리 높이가 항상 일정 (skew가 없음)
  - 최소 저장공간 활용 비율 보장 가능 (예: 50%는 최소 사용)
  - Leaf 노드 데이터는 키에 의하여 정렬됨

B+-Tree의 예

틀린 b+ tree구조(hyori때문에) 참고X할 것



Title	Artist	Genre	Year	Rate
I Know	SEO	2	1992	4.2
Fly High	EpicHigh	2	2005	3.5
Tonight	CLON	1	1998	2.7
10 Minutes	Hyori	3	2004	5.0
…	…	…	…	…
Color Your Soul	Classiqui	4	2005	4.1

MUSIC Table

index는 성능을 위해 존재하므로 최적의 index만 살아남는다.

Syntax

- CREATE [UNIQUE] INDEX *index_name* ON *table_name* (*column_list* …);

```
CREATE INDEX idx_emp_ename ON emp(ename);
```

- DROP INDEX *index_name*;

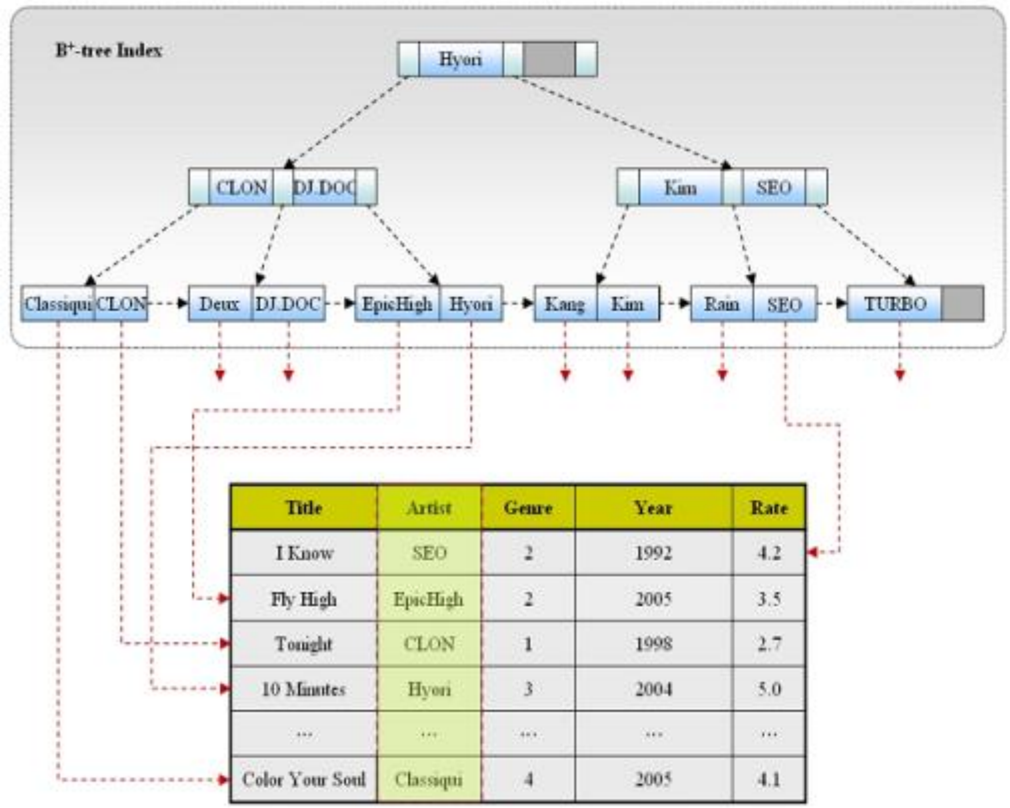
```
DROP INDEX idx_emp_ename;
```

- Dictionary
  - USER_INDEXES
  - USER_IND_COLUMNS

```
SELECT t.index_name, t.uniqueness,
       c.column_name, c.column_position
FROM user_indexes t, user_ind_columns c
WHERE c.index_name = t.index_name AND t.table_name = 'EMP';
```

Oracle의 데이터 접근 방법

- Full Table Scan
  - 전체 테이블 순차적 접근
- Index Scan
  - 인덱스를 통한 접근
  - Index Unique
  - Index Range
  - Index Full Scan



Title	Artist	Genre	Year	Rate
I Know	SEO	2	1992	4.2
Fly High	EpicHigh	2	2005	3.5
Tonight	CLON	1	1998	2.7
10 Minutes	Hyori	3	2004	5.0
…	…	…	…	…
Color Your Soul	Classiqui	4	2005	4.1

MUSIC Table

MYONGJI UNIVERSITY

Query Optimizer

- Query Optimizer가 주어진 질의의 처리방법 결정
  - Cost-Based Optimization
  - Heuristic-Based Optimization

- 일반적인 고려 요소
  - 인덱스의 유무
  - 테이블의 크기
  - 데이터의 분포
  - …

Composite Index & Covering Index

- **Composite Index**
  - 둘 이상의 컬럼의 쌍에 대한 인덱스. **순서 중요!**
  - 예)

```
CREATE INDEX idx_emp1 ON emp(dept,name);
```

ex.
A-Kim
A-Lee
A-Mark
....
B-Adam
...

    - SELECT * FROM emp WHERE **dept = 'A'** AND **name = 'B'**
    - SELECT * FROM emp WHERE **dept = 'A'**
    - SELECT * FROM emp WHERE **name = 'B'**

3번째인 name ='B'는 index도움을 못받음(의미가 없다.) - 테이블 스캔이랑 같음.

- **Covering Index** (covered query)
  - Table 참조 없이 Index만으로 처리 가능한 경우
  - 일반적으로 성능이 우수
  - 예)
    - SELECT dept, name FROM emp WHERE dept='AAA';
    - SELECT COUNT(*) FROM emp WHERE dept='AAA';

Hint Index

- 강제로 질의에서 특정 인덱스를 사용하도록 강요함
- Oracle 예)

```
SELECT /*+ index (employees EMP_EMAIL_UK) */ email
FROM employees;
```

Other types of indices

- **Indices supported by Oracle**
  - Create index syntax creates **B+-Tree** index
  - **Bitmap index**

```
create bitmap index bitmap_index on branch(state);
```

cf. bitmap index는 state한개만 즉 합성 불가

  - Reverse Key Indices
  - Function-based Indices

Reverse Key Indices

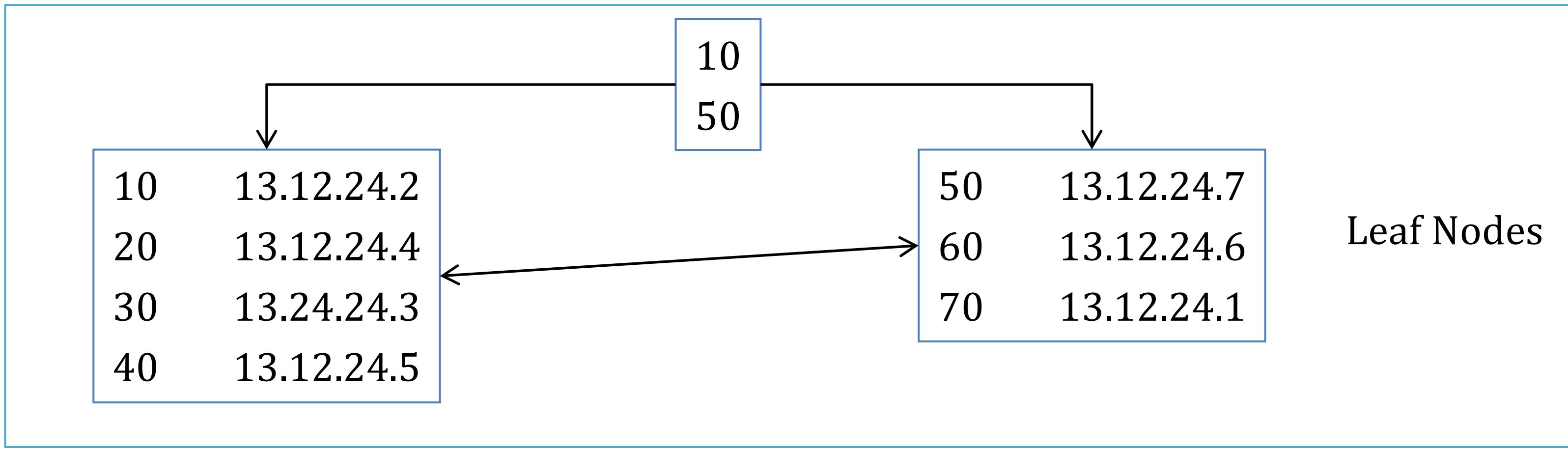
- **Indexes on column have another major performance problem**
  - frequently, the primary key of a table is an *auto-incrementing* sequence number

EMPNO	NAME	DEPT	SAL
70	Bob	10	450
10	Frank	10	550
30	Ed	30	575
20	Adam	20	345
40	David	10	550
60	Graham	30	625
50	Charles	20	330

  - *EMPNO* is a monotonically incrementing sequence number

Reverse Key Indices

- **A normal B+-Tree index**



- **Advantages**
  - as we employ new people and perform inserts on the table, it should be evident that we are in absolutely no danger of ever needing to revisit an earlier leaf node
  - never going to experience a *block split* amongst the leaf nodes (except the last one)

Reverse Key Indices

- **Disadvantages**
  - every new index entry will always take place on the last leaf node!

  → if 100 users simultaneously insert a new record into the table, they will all be fighting for access to the same leaf node (especially in a parallel server environment)

  → directly manifests itself in extremely poor response times for users attempting to perform a simple bit of DML

Reverse Key Index

- **Reverse Key Index**
  - *reverse* the key value and use it as an index key value

    거꾸로 하기
  - mechanism that would *scatter* inserts randomly across the base of the index
  - just an ordinary B+-Tree index
    - if you enter a new record in the table with a sequence number of say, 7891, we index it as 1987
    - if you enter sequence number 7982, we index it as 2897
    - 7901 gets indexed as 1097, and so on

Reverse Key Index

- **Reverse Key Index**
  - advantage
    - the contention issue thus disappears

      100개 한번에 들어오때 락걸리는 것을 방지가능
  - disadvantage
    - you are now scattering new inserts back and forth across the base of the index
    - inserting new leaf node entries amongst existing entries, which means that block splits are a possibility once again
  - Syntax

```
CREATE INDEX emp_empno_pk ON emp(empno) reverse;
```

Function-based Indices

- **Problems**
  - data stored by DBMS are *sensitive to case*, while standard SQL is not!
    - if in a table, we store people's names as Fred and Bob, then a search for FRED, BOB, fred, or bob will turn up no rows

      저장되는 값들은 반드시 대소문자 구별!

- **Solution:**

```
select name, salary, dept from emp
    where upper(name) = 'BOB';
```

대소문자 구별없이 다 같은취급

  - will *not* use the index on the NAME column even if there is an index on it (because of the **`upper()`** function)
    - perform full table scan & apply **`upper()`** function for each row encountered

MYONGJI UNIVERSITY

Function-based Indices

- **Better solution:** 즉 인덱스에선 같은 취급, 실제 데이터 다른 취급
  - create a *function-based* index

    `create index upper_name_idx on emp(upper(name));`

  - now, when we issue the query, we are likely to be able to make use of the index

  - the WHERE predicate of the query does not need to be computed for each record (`where upper(name) = 'BOB';`)

MYONGJI UNIVERSITY

Summary

- Index
  - 테이블 검색을 빠르게 하기 위하여 미리 구축해 두는 저장 구조
  - 종류: Tree, Bitmap ...
- Index 장점
  - 대량의 데이터에서 특별한 값을 빠르게 검색
  - 검색, 조인, 정렬...
- Index 단점
  - 유지 비용 (데이터 변경 시 인덱스도 변경하여야 함)
  - 저장 공간
- When to use?
  - WHERE 절의 조건이나 Join에 자주 사용되는 컬럼
  - 매우 큰 테이블에서 2~4% 레코드만 선택될 때
  - 값의 종류가 다양할 때 (예. 이름 vs. 성별?)
  - 자주 변경되지 않을 때

THE END