UST 과학기술연합대학원대학교
UNIVERSITY OF SCIENCE & TECHNOLOGY

Lecture 8:
Index Mechanisms

Dr. Kyong-Ha Lee
(kyongha@kisti.re.kr)

KiSTi www.kisti.re.kr 한국과학기술정보연구원
Korea Institute of Science and Technology Information

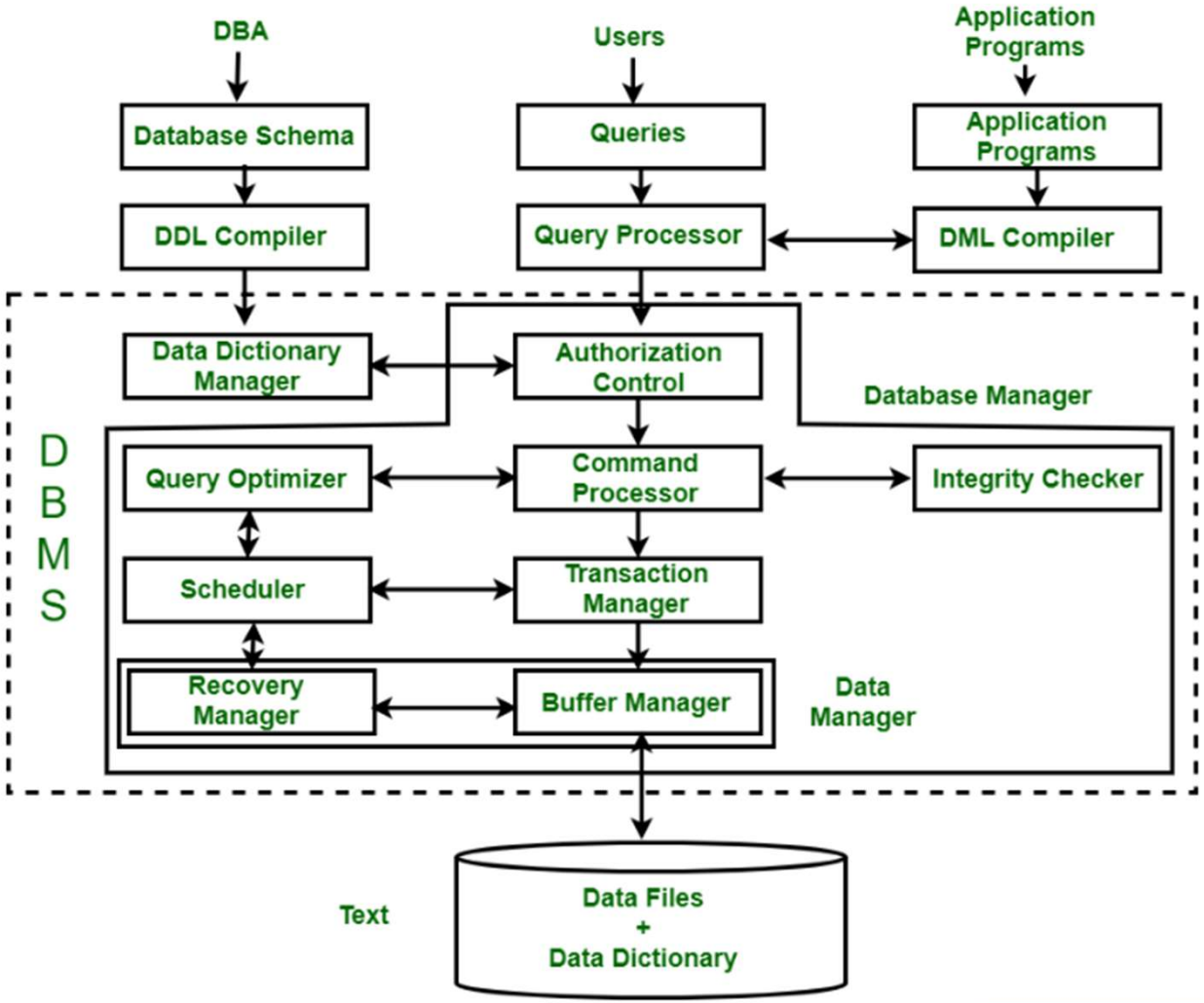
Contents

Brief overview of this lecture

- Basic theories and principles about Index mechanisms used in database systems
- Not much discussion on implementation or tools, but will be happy to discuss them if there are any questions

*Disclaimer: these slides are based on the slides created by the authors of Database System Concepts 7th ed. and modified by K.H. Lee.

DBMS Architecture Overview



* Source: Structure of Database Management System - GeeksforGeeks

Basic Concepts

- Indices are used to speed up accesses to desired data
- **Search Key** – attribute to set of attributes used to look up records in a file
- An **index file** consists of records (called **index entries**) of the form

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

Index Evaluation metrics

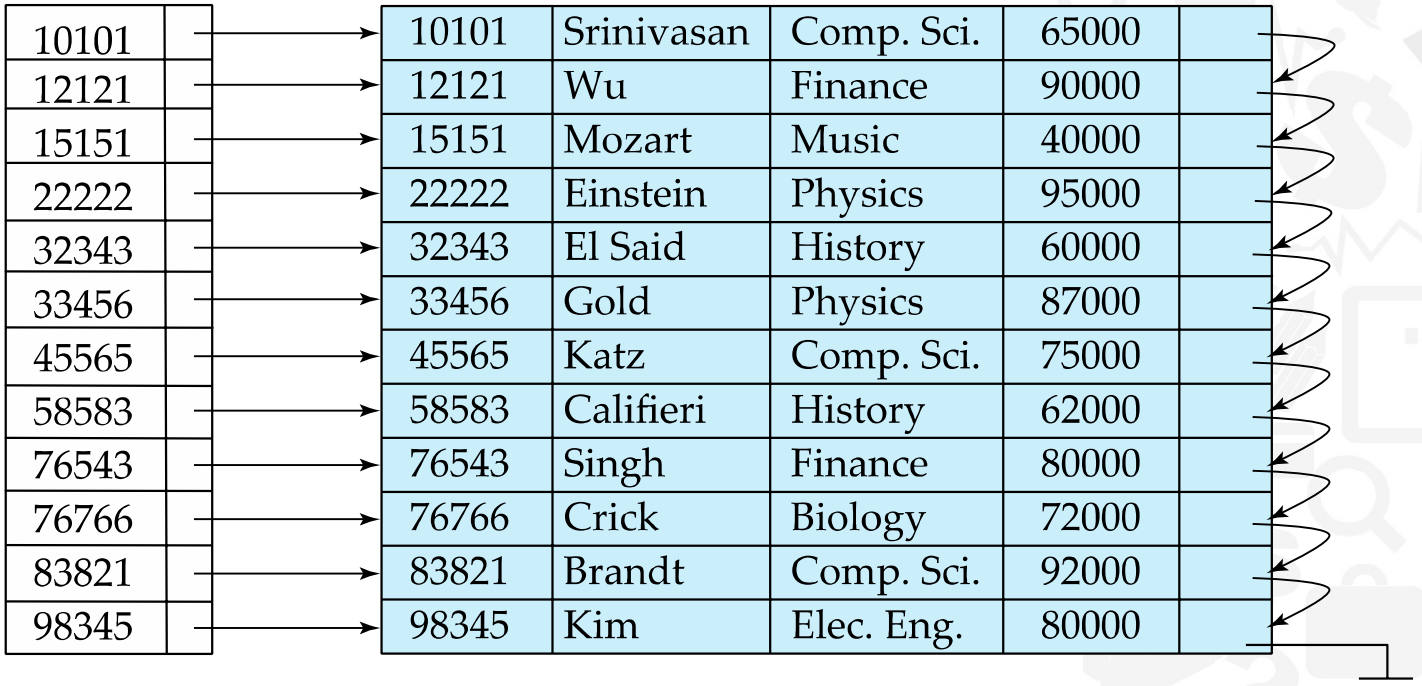
- Access type supported
  - e.g., Records with a specified value in the attribute
  - Records with an attribute value in a specfic range of values
- Search time
  - $O(N) > O(\log_2 N) > O(\log_m N) > O(1)$
- Insertition time
- Deletion time
- Space overhead required for maintaining the index
  - (major premise) Index is much smaller than real data

Ordered Indicies

- Index entries are stored in a sorted order of search key values
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **primary index**
  - The search key of a primary index is usually but not necessarily the primary key
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file
  - Also called **nonclustering index.**
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.

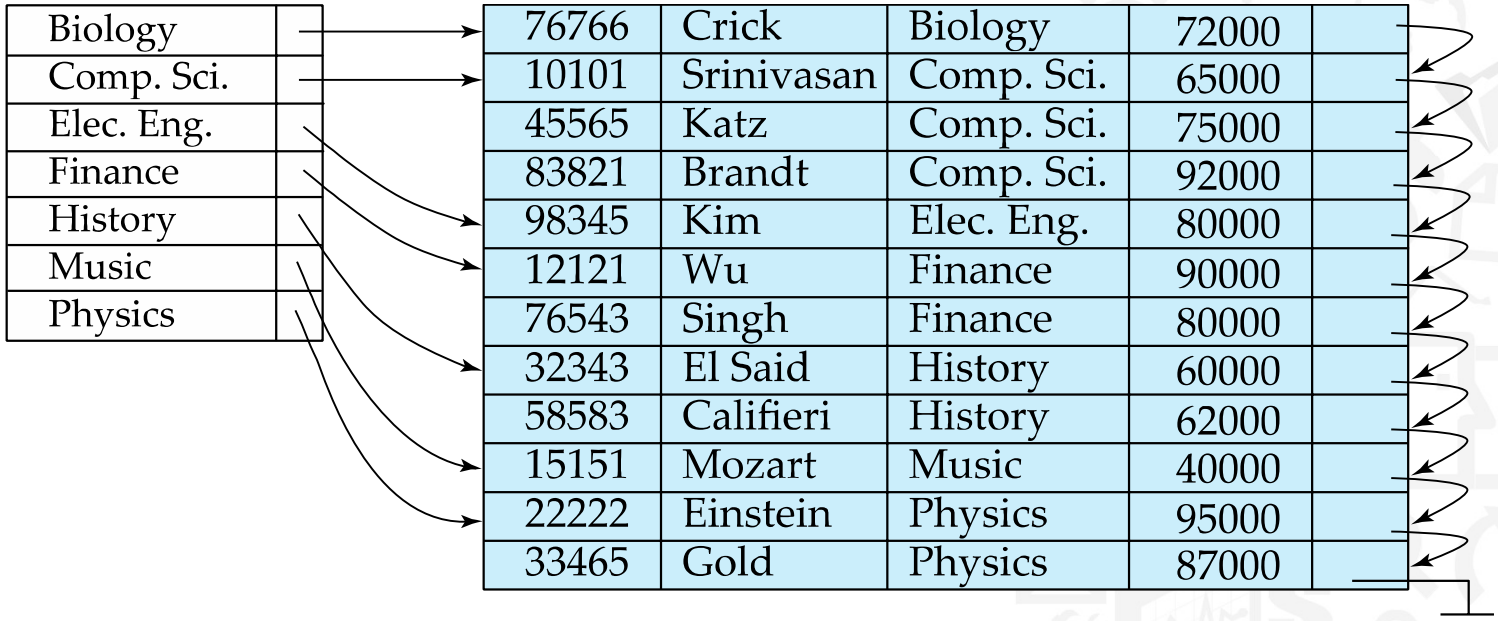
Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



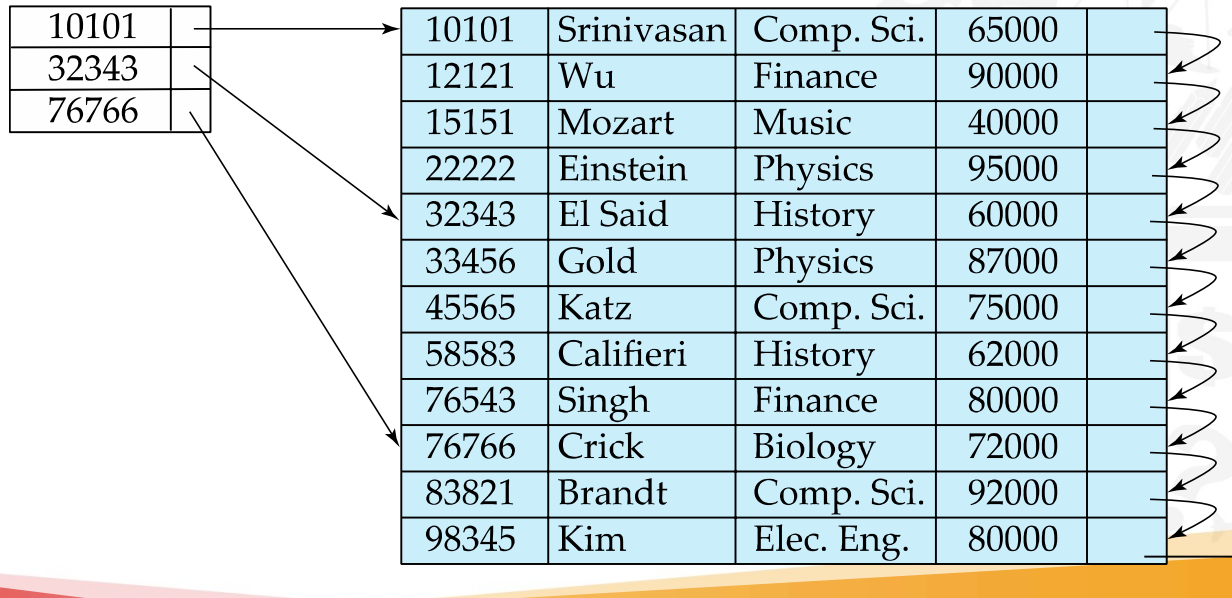
Dense Index Files (Cont'd)

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



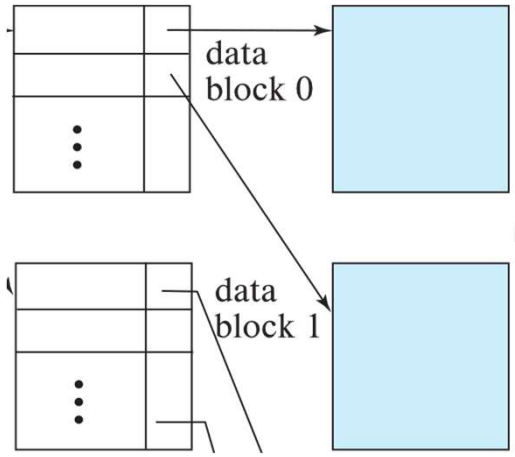
Sparse Index Files

- Sparse Index: contains index records for only some search-keys
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points



Sparse Index Files (Cont'd)

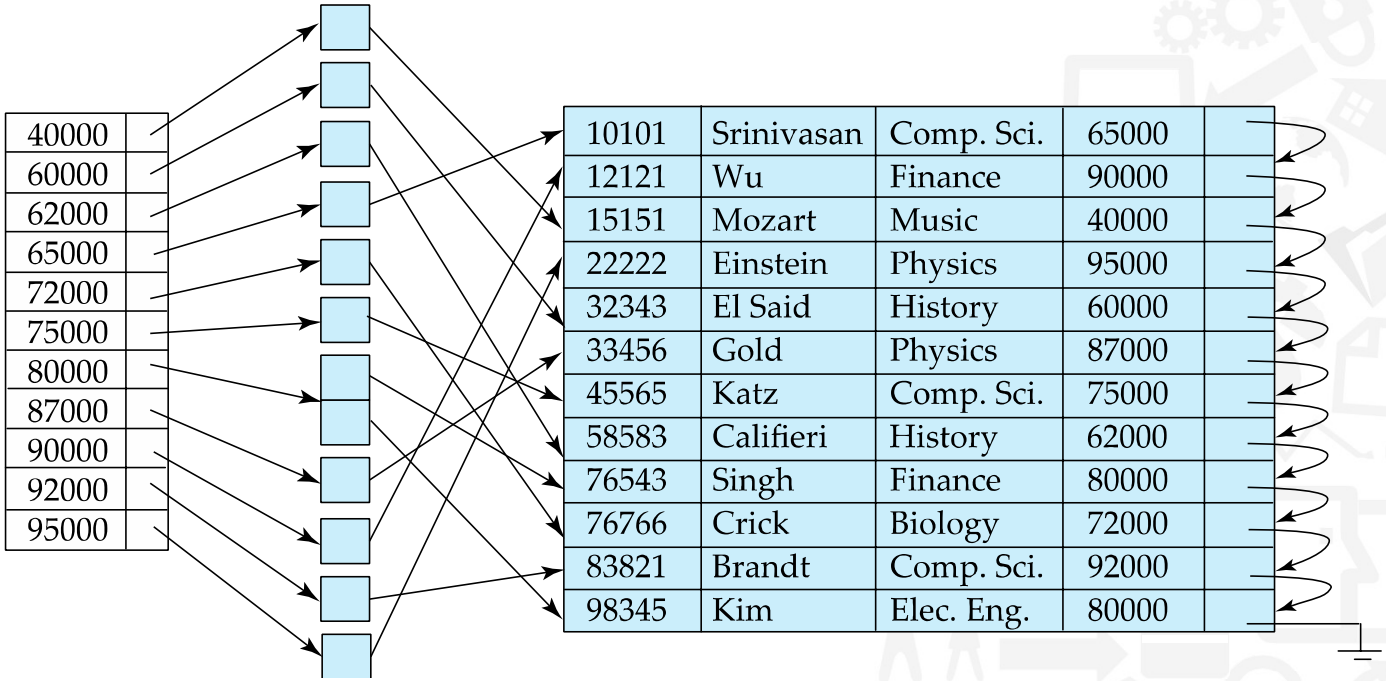
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**:
  - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key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



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

Secondary indices example

- Secondary index on salary field of instructor



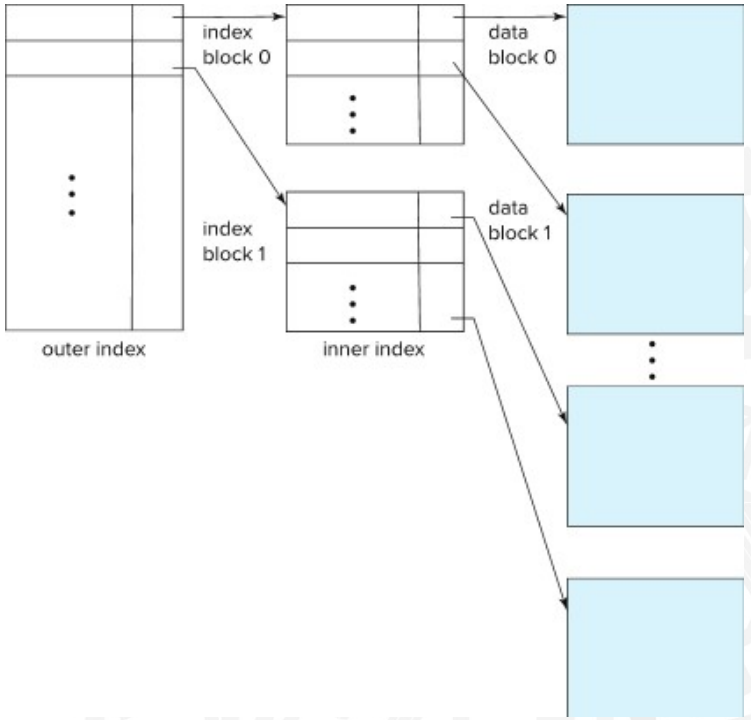
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Clustering vs. Non-clustering Indices

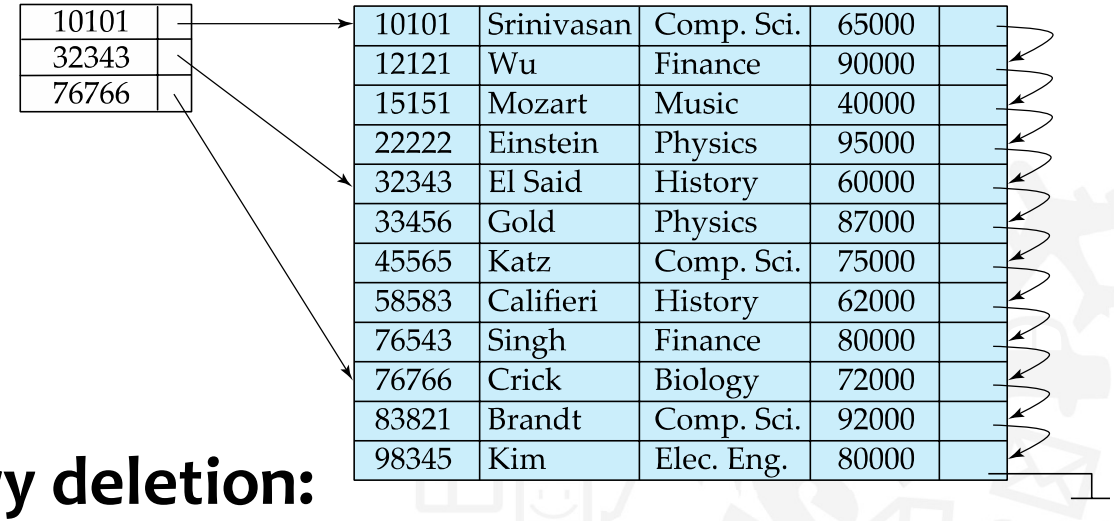
- Indices offer substantial benefits when searching for records.
- BUT: **indices imposes overhead on database modification**
  - when a record is inserted or deleted, every index on the relation must be updated
  - When a record is updated, any index on an updated attribute must be updated
- Sequential scan using clustering index is efficient, but a sequential scan using a secondary (nonclustering) index is expensive on magnetic disk
  - Each record access may fetch a new block from disk
  - Each block fetch on magnetic disk requires about 5 to 10 milliseconds

Multi-level Index

- 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 the 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.



Index Update : Deletion



- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- **Single-level index entry deletion:**
  - **Dense indices** – deletion of search-key is similar to file record deletion.
  - **Sparse indices**
    - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value of the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it
    - Need to create space for new entry, overflow blocks may be required
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

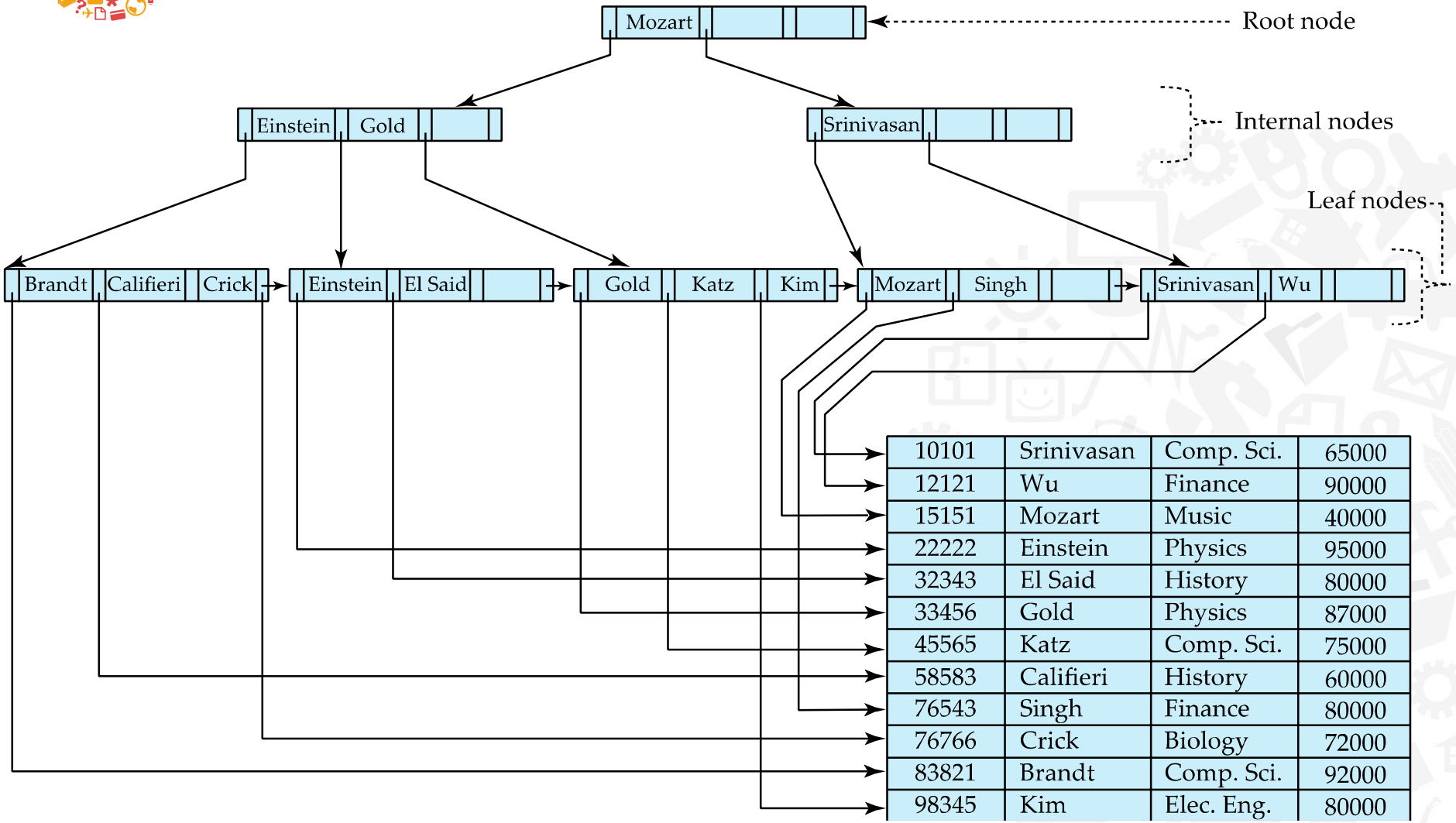
Indices on Multiple Keys

- **Composite search key**
  - E.g., index on *instructor* relation on attributes (*name, ID*)
  - Values are sorted lexicographically
    - E.g. (John, 12121) < (John, 13514) and
      (John, 13514) < (Peter, 11223)
  - Can query on just *name*, or on (*name, ID*)

B^+-Tree Index

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, 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.
  - 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+-Tree Index Example



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

- A rooted tree satisfying the following properties
  - All root-to-leaf paths are of the same length
  - 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.
- Typical node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

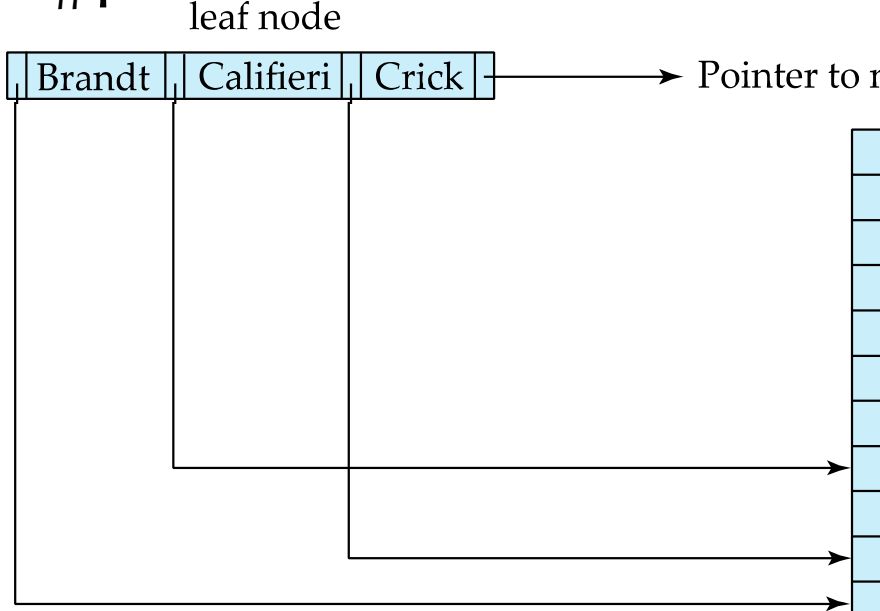
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+-Tree

- For $i = 1, 2, \ldots, n-1$, 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

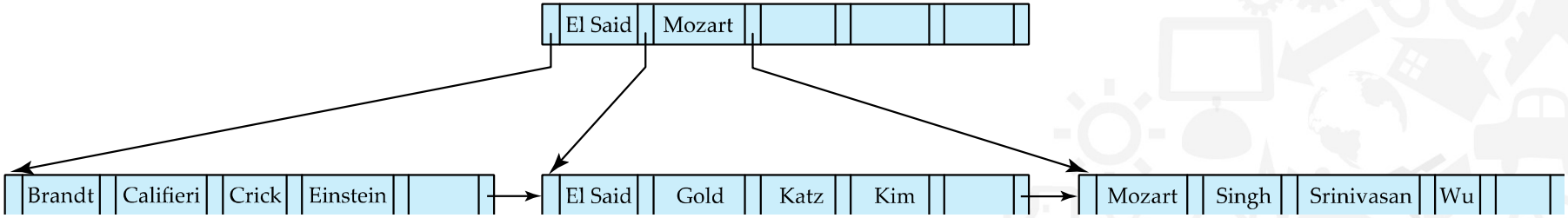
Non-Leaf Nodes in B$^+$-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree 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 subtree to which P_n points have values greater than or equal to K_{n-1}
  - General structure

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

Example of B+-tree

- B+-tree (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

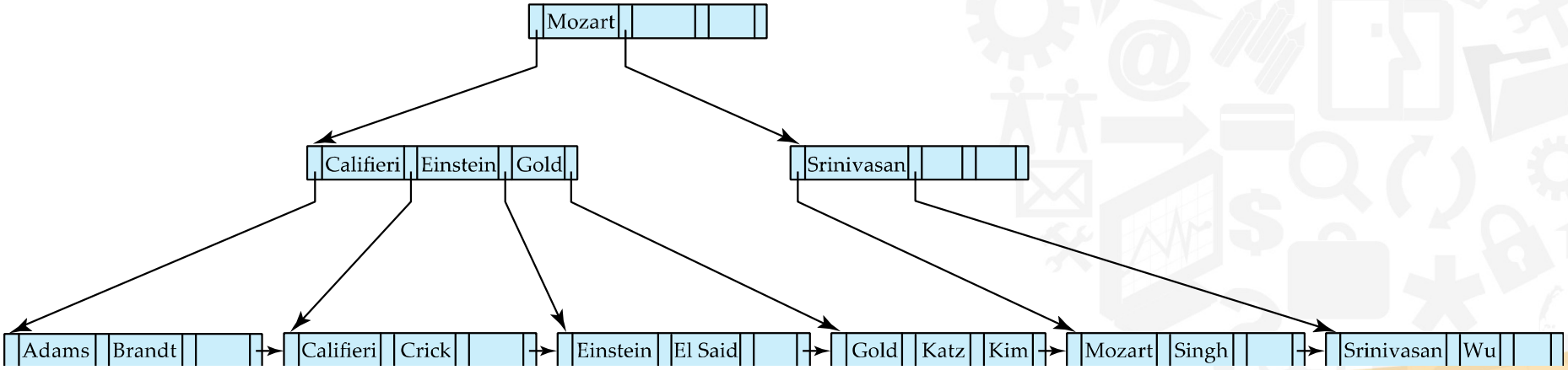
Observations

- 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 $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$**
- Insertions and deletions to the main file can be handled efficiently, as **the index can be restructured in logarithmic time**

Queries on B$^+$-tree

function *find*(*v*)

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. */



Queries on B+-Trees (Cont'd)

- If there are *K* search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- 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 traversal from root to leaf.
- 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

Non-Unique Keys

- If a search key a_i is not unique, create instead an index on a composite key (a_i, A_p), which is unique
  - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range $(v, -\infty)$ to $(v, +\infty)$
- But more I/O operations are needed to fetch the actual records
  - If the index is clustering, all accesses are sequential
  - If the index is non-clustering, each record access may need an I/O operation

Updates on B$^+$-Trees: Insertion

Assume record already added to the file. Let

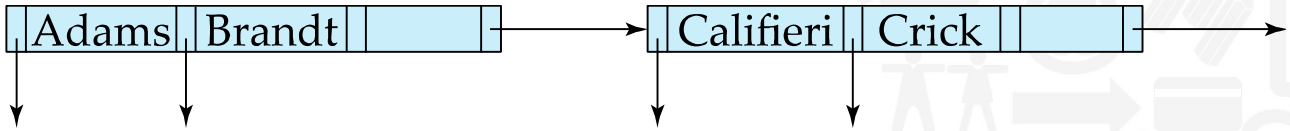
pr be pointer to the record, and let

v be the search key value of the record

1. Find the leaf node in which the search-key value would appear
    1. If there is room in the leaf node, insert (v, *pr*) pair in the leaf node
    2. Otherwise, split the node (along with the new (*v*, *pr*) entry) as discussed in the next slide, and propagate updates to parent nodes.

Updates on B+-Trees: Insertion (Cont'd)

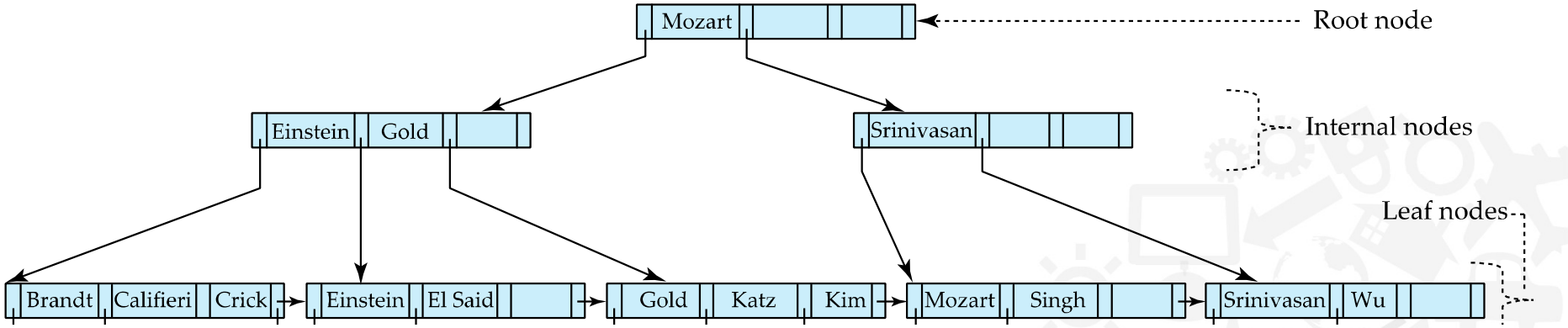
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p*, and let *k* be the least key value in *p*. Insert (*k*,*p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.

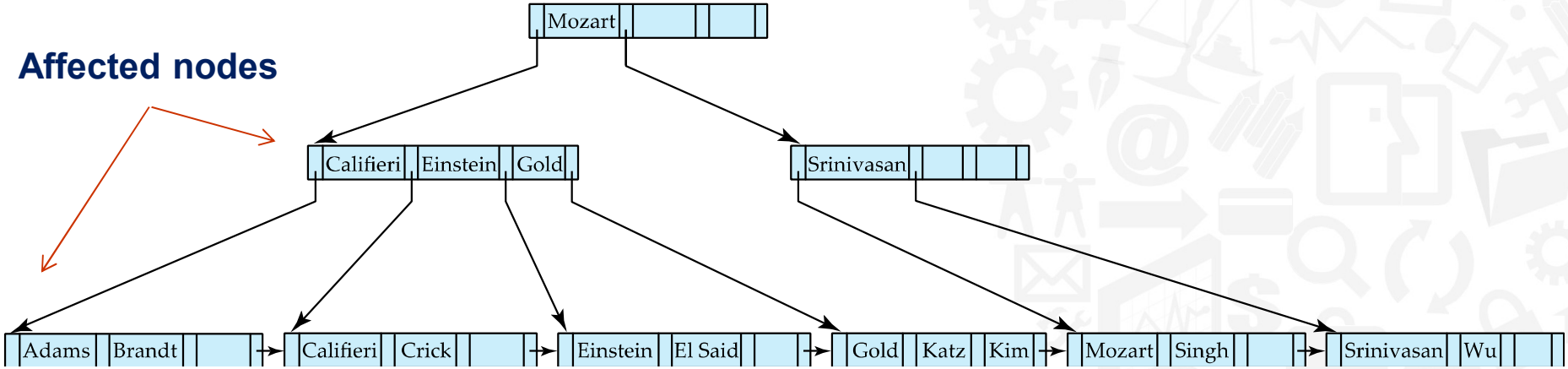


Result of splitting node containing Brandt, Califieri and Crick on inserting Adams

Next step: insert entry with (Califieri, pointer-to-new-node) into parent

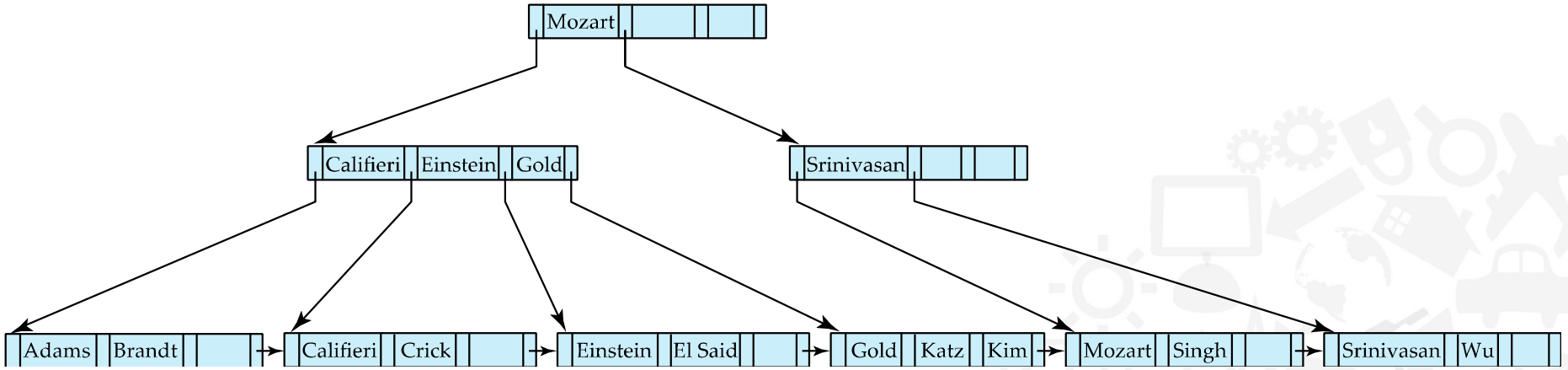
B+-Tree Insertion



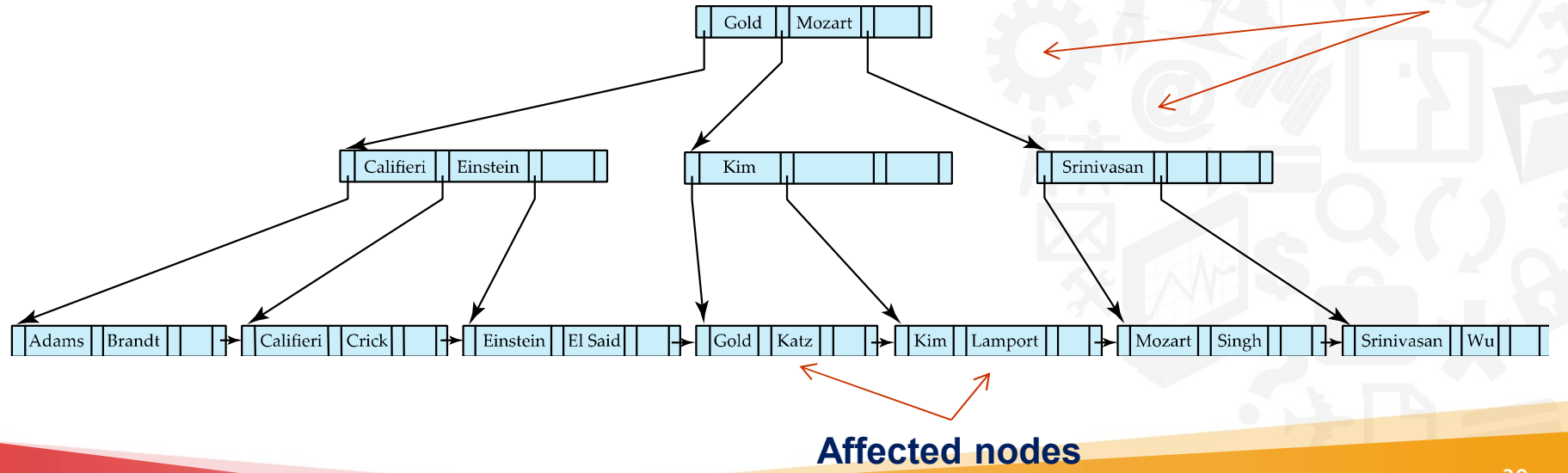


B+-Tree before and after insertion of “Adams”

B⁺-Tree Insertion

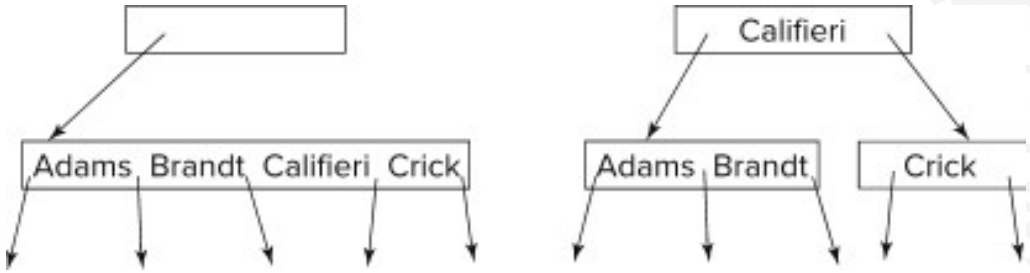


B⁺-Tree before and after insertion of "Lamport"



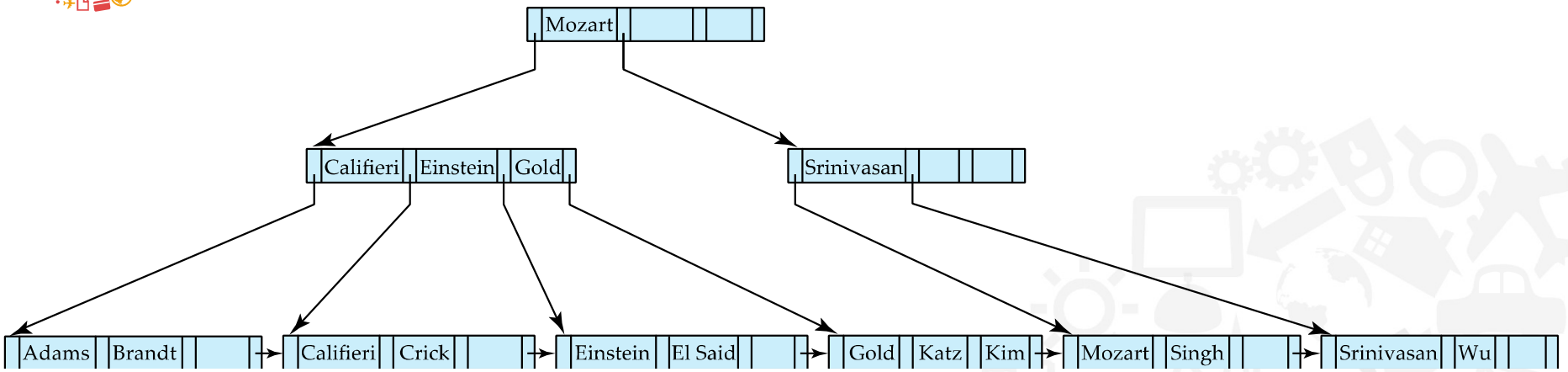
B+-Tree Insertion (Cont.)

- 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 $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N
- Example

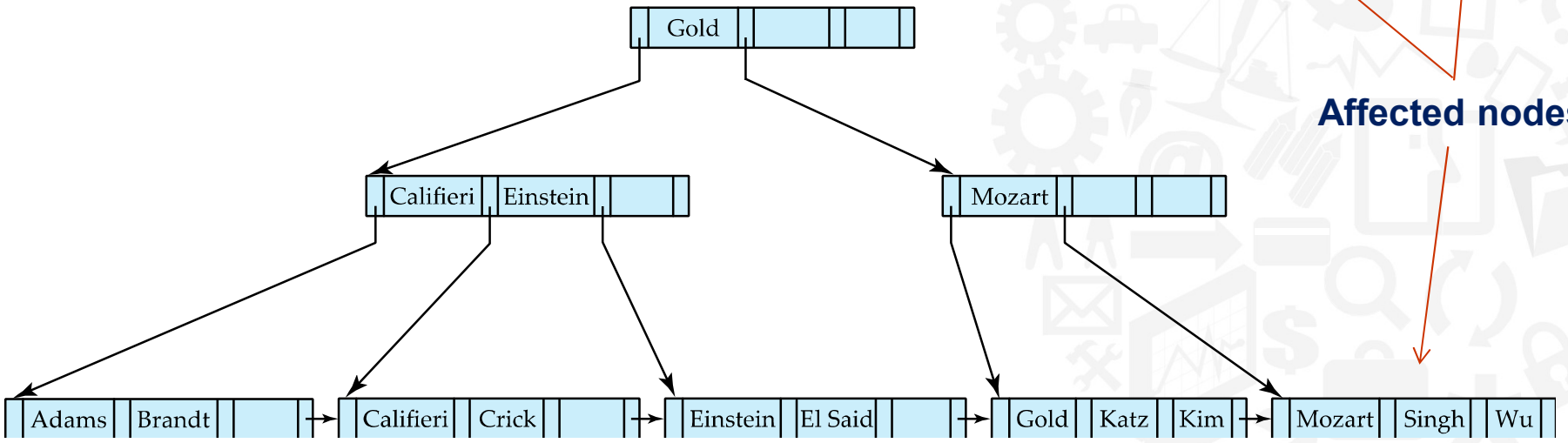


- **Read pseudocode in book!**

B+-Tree Deletion



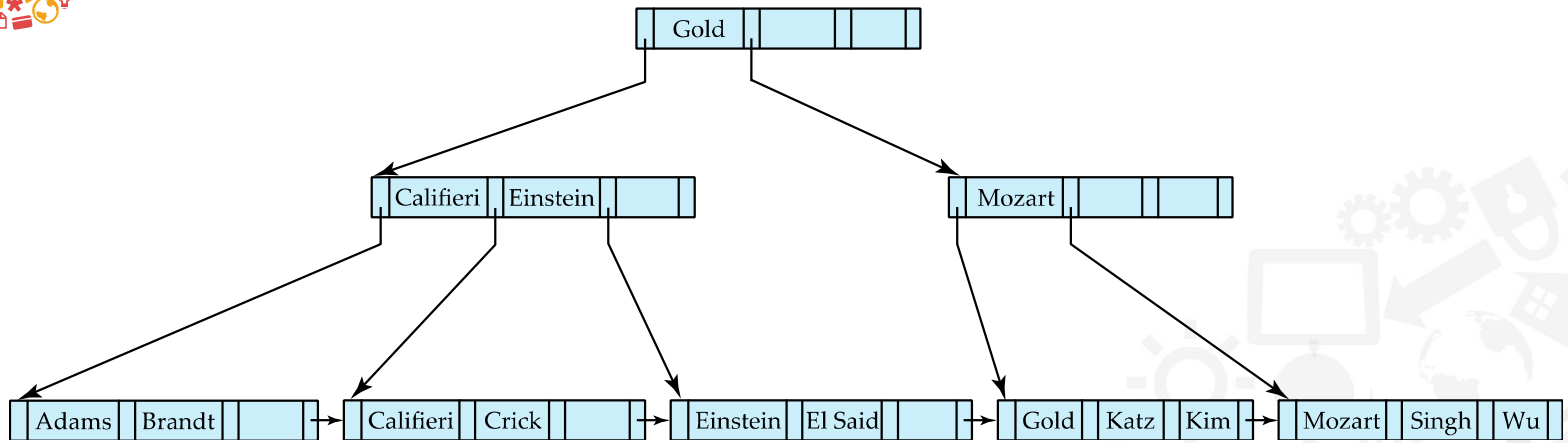
Before and after deleting "Srinivasan"



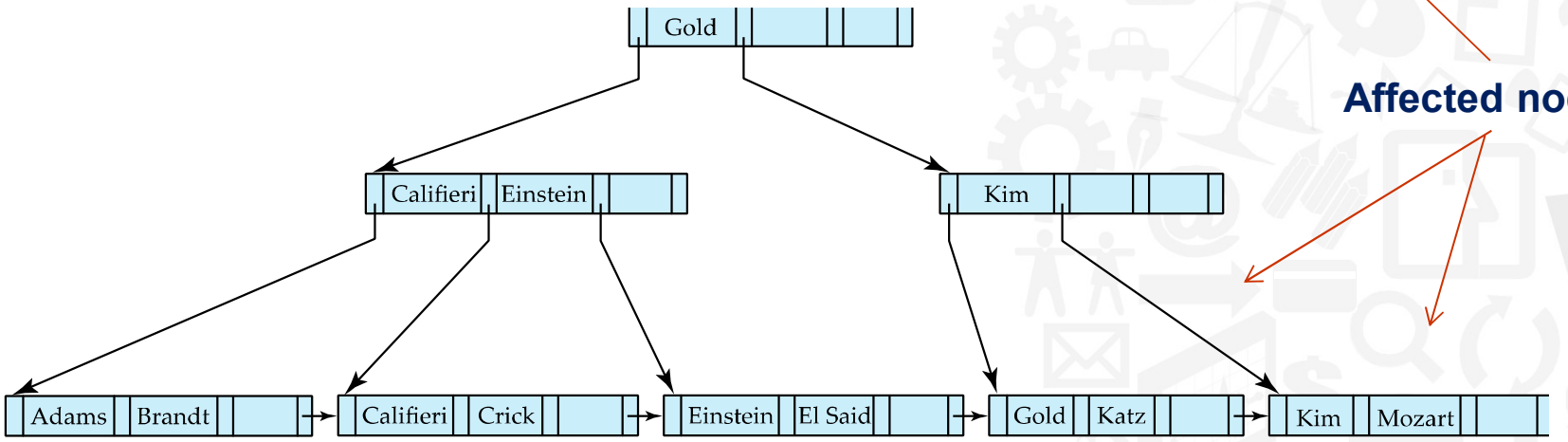
Affected nodes

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

B+-Tree Deletion(Cont'd)

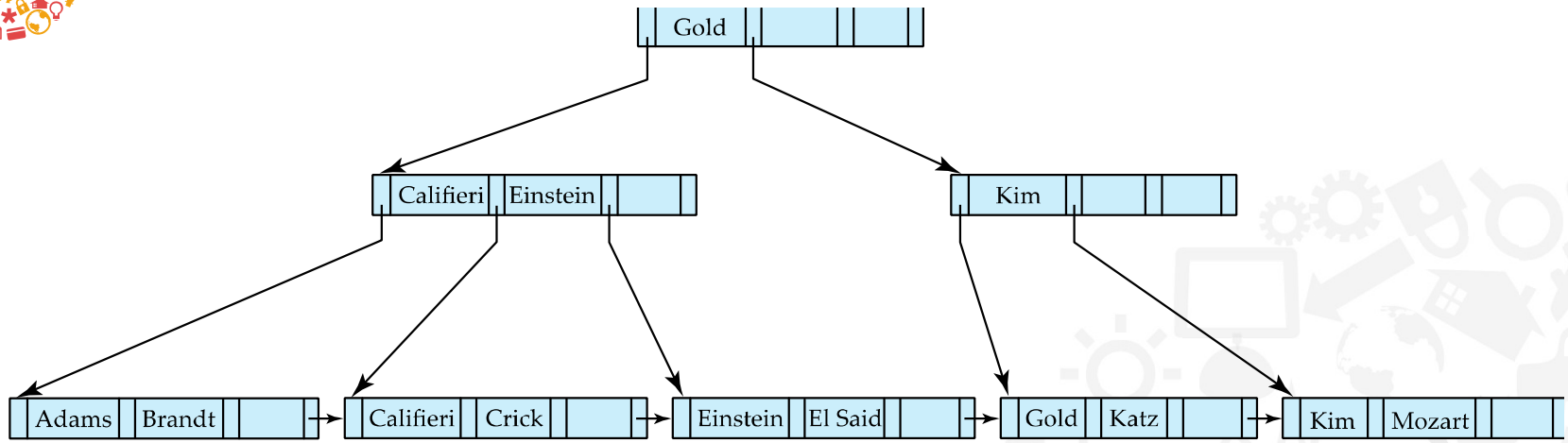


Before and after deleting "Singh" and "Wu"

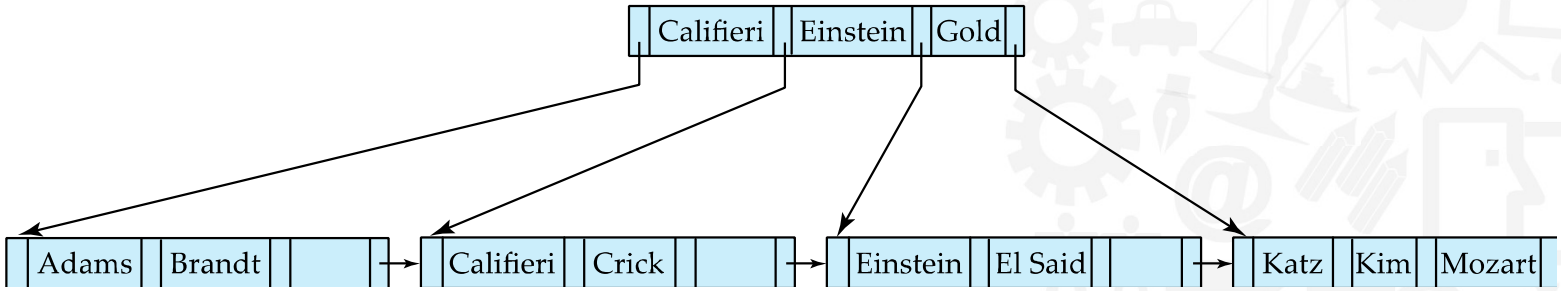


- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result

B+-Tree Deletion(Cont'd)



Before and after deletion of "Gold"



- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

B+-Tree Deletion(Cont'd)

Let *V* be the search key value of the record, and *Pr* be the pointer to the record.

- Remove (*Pr*, *V*) from the leaf node
- If the node has too few entries and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.

Complexity of Updates

- Cost (in terms of # of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout of n, worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, # of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

Other Issues in Indexing

- **Record relocation and secondary indices**
  - If a record moves, all secondary indices that store record pointers have to be updated
  - Node splits in B^+-tree file organizations become very expensive
  - *Solution*: use search key of B^+-tree file organization instead of record pointer in secondary index
    - Add record-id if B^+-tree file organization search key is non-unique
    - Extra traversal of file organization to locate record
      - Higher cost for queries, but node splits are cheap

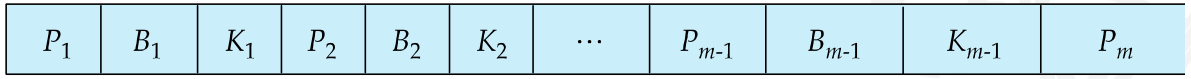
Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g., "Silas" and "Silberschatz" can be separated by "Silb"
  - Keys in leaf node can be compressed by sharing common prefixes

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 nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

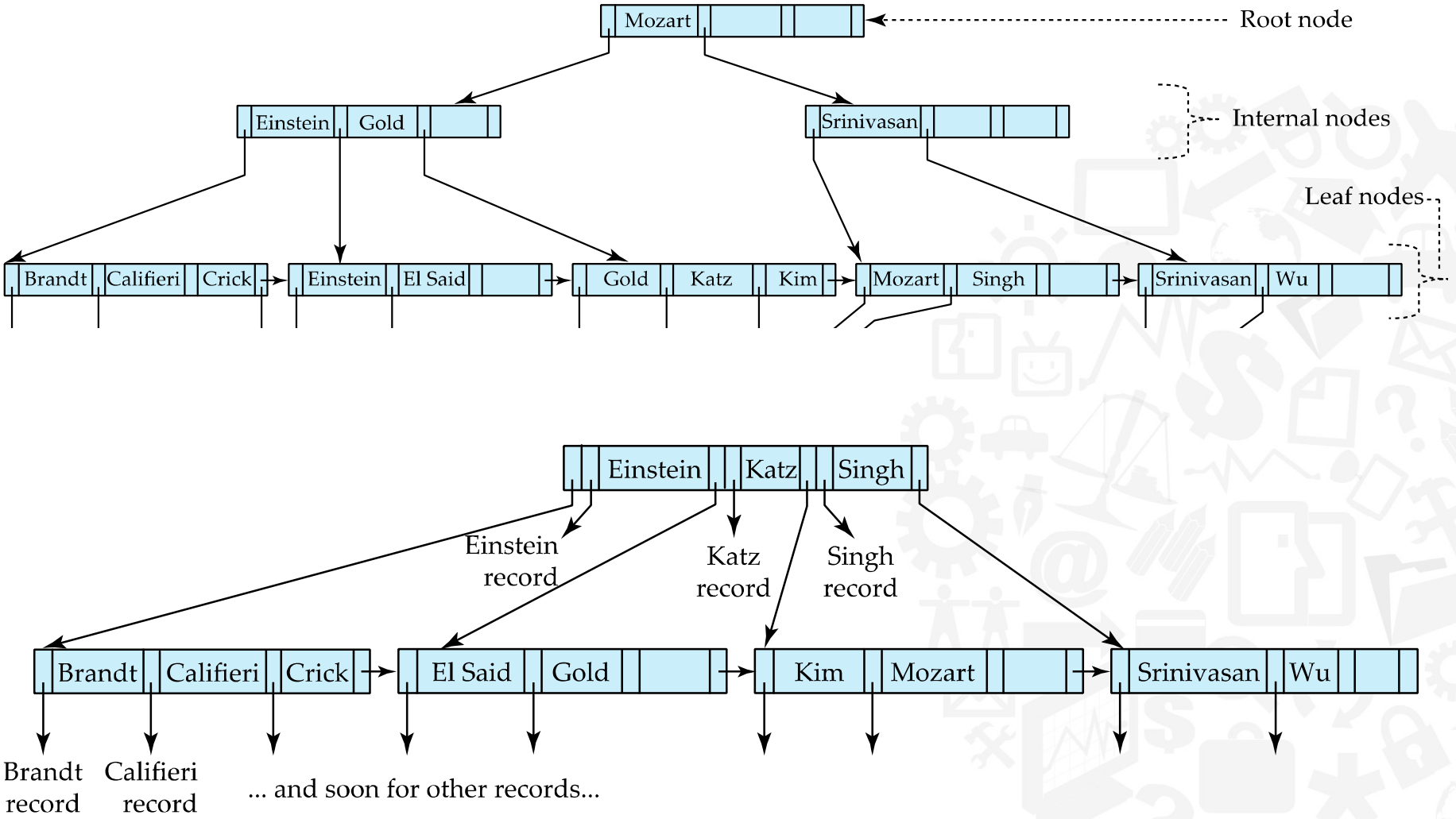
(a)



(b)

- Nonleaf node – pointers Bi are the bucket or file record pointers.

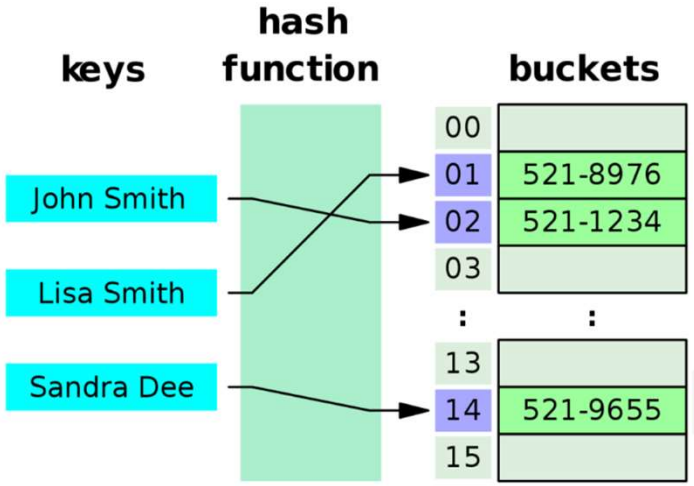
B⁺-tree vs. B-tree



Pros and Cons in B-Tree

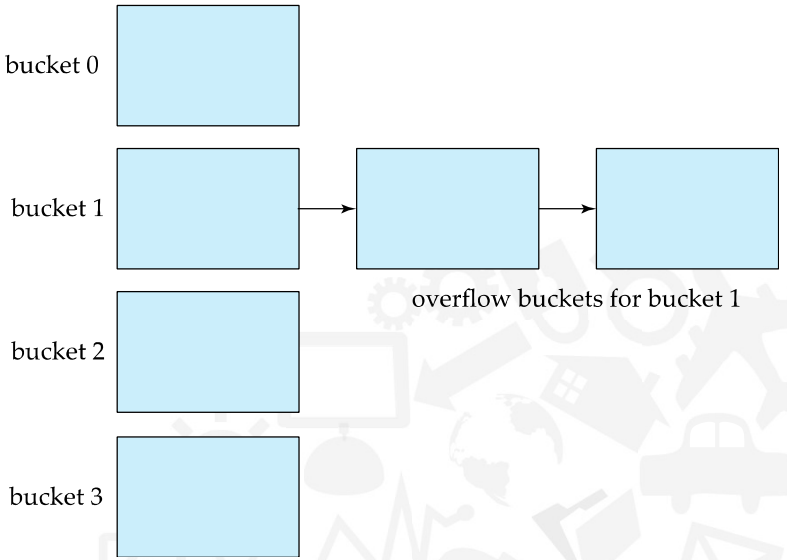
- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes p**ossible to find search-key value before reaching leaf node.**
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - **Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree**
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, **advantages of B-Trees do not out weigh disadvantages**.

Static Hashing



- A bucket is a unit of storage containing 1+ entries
  - Typically a disk block
- Hash function ***h*** is a function from the set of all search-key values ***K*** to the set of all bucket addresses ***B***
  - ***h*** is used to locate entries for all accesses(insertion, deletion)
  - Entries with different search-keys can be mapped to the same bucket
    - Entire bucket has to be scanned sequentially to locate an entry
- **Hash index** stores entries with pointers to records in buckets
- In **Hash file organization**, buckets store records

Bucket Overflow



- Reasons
  - Insufficient buckets
  - Skew in distribution of records
    - Multiple records have the same search-key value
    - Hash function h produces non-uniform distribution of key values
- Eventually, bucket overflow may happen. So we handle this by using **overflow buckets**
- **Overflow chaining** – overflow buckets are chained in a linked list
  - Called Closed addressing or open hashing

Deficiencies of Static Hashing

- In static hashing, function ***h*** maps search-key values to a fixed set of ***B*** of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: **allow the number of buckets to be modified dynamically**

Dynamic Hashing

- Periodic rehashing
  - Create new hash table of size 2 times the size of the prev. hash table
  - Rehash all entries to new table
- Linear Hashing
  - Do rehashing in an incremental manner
- Extendable hashing
  - Tailored to disk-based hashing, w/ buckets shared by multiple hash values
  - Doubling of # of entries in hash table, w/o doubling # of buckets

Comparision of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key**ordered indices are to be preferred**
  - **If range queries are common**, In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees
  - MariaDB support B or R or HASH depending on the underlying storage engines

Storage Engine	Permitted Indexes
Aria	BTREE, RTREE
MyISAM	BTREE, RTREE
InnoDB	BTREE
MEMORY/HEAP	HASH, BTREE

*source: Storage Engine Index Types - MariaDB Knowledge Base

Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

  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.

Other Features

- **Covering indices**
  - Add extra attributes to index so (some) queries can avoid fetching the actual records
  - Store extra attributes only at leaf
    - Why?
- Particularly useful for secondary indices
  - Why?

Index Creation

- Example
  create index *takes_pk* **on** *takes* (*ID,course_ID, year, semester, section*)
  drop index *takes_pk*
- Most database systems allow specification of type of index, and clustering.
- Indices on primary key created automatically by all databases
  - Why?
- Some database also create indices on foreign key attributes
  - Why might such an index be useful for this query:
    - $takes \bowtie \sigma_{name='Shankar'}(student)$
- Indices can greatly speed up lookups, but impose cost on updates
  - Index tuning assistants/wizards supported on several databases to help choose indices, based on query and update workload

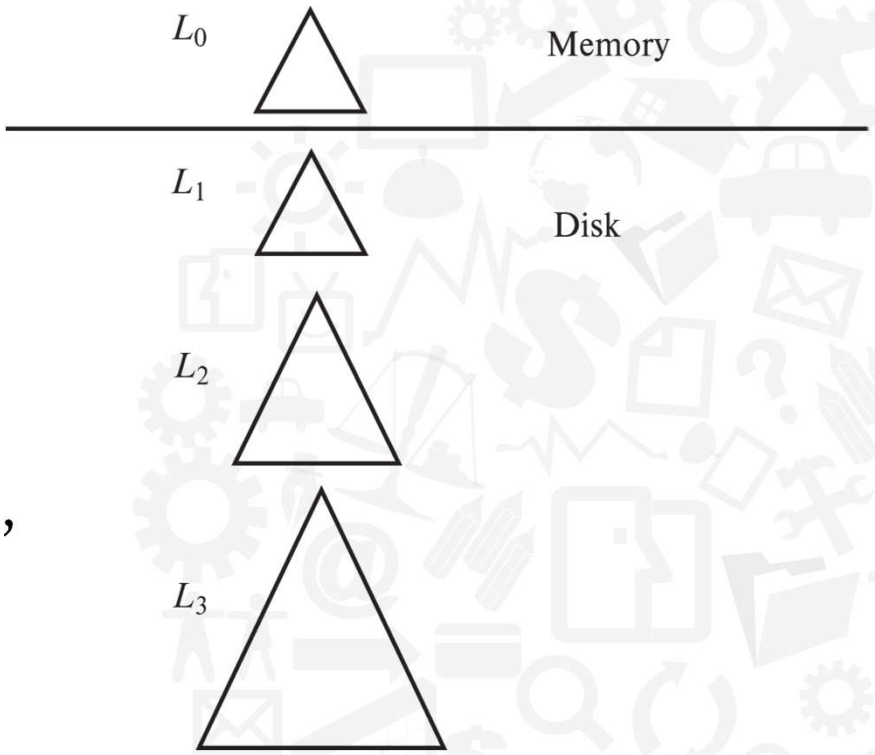
Write-Optimized indicies

- Performance of B^+-trees can be poor for write-intensive workloads
  - One I/O per leaf, assuming all internal nodes are in memory

- Two approaches to reducing cost of writes
  - **Log-structured merge tree(LSM tree)**
  - **Buffer tree**

Log Structured Merge(LSM) Tree

- Consider only inserts/queries for now
- Records inserted first into in-memory tree (L_0 tree)
- When in-memory tree is full, records are purged to disk (L_1 tree)
  - B^+-tree constructed using bottom-up build by merging existing L_1 tree with records from L_0 tree
- When L_1 tree exceeds some threshold, merge into L_2 tree
  - And so on for more levels
  - Size threshold for L_{i+1} tree is *k* times size threshold for L_i tree



LSM Tree(Cont'd)

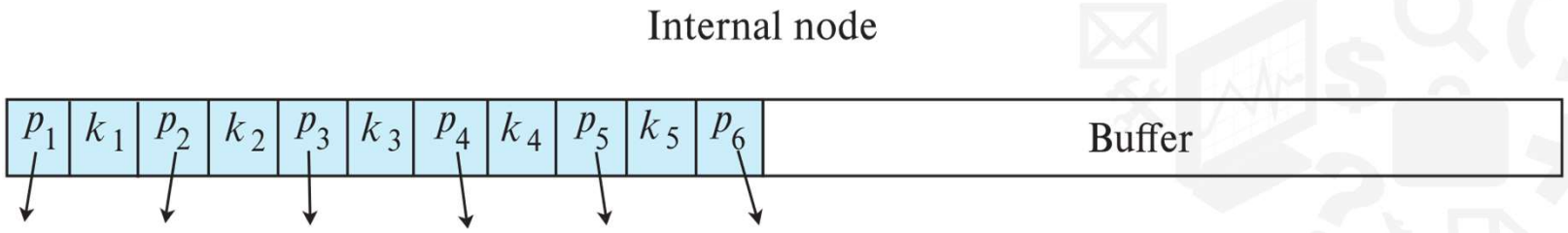
- Benefits of LSM approach
  - Inserts are done using only sequential I/O operations
  - Leaves are full, avoiding space wastage
  - Reduced number of I/O operations per record inserted as compared to normal B^+-tree (up to some size)
- Drawback of LSM approach
  - Queries have to search multiple trees
  - Entire content of each level copied multiple times
- Stepped-merge index
  - Variant of LSM tree with multiple trees at each level
  - Reduces write cost compared to LSM tree
  - But queries are even more expensive
    - Bloom filters to avoid lookups in most trees
- Details are covered in Chapter 24

LSM Tree(Cont'd)

- Deletion handled by adding special "delete" entries
  - Lookups will find both original entry and the delete entry, and must return only those entries that do not have matching delete entry
  - When trees are merged, if we find a delete entry matching an original entry, both are dropped.
- Update handled using insert+delete
- LSM trees were introduced for disk-based indices
  - But useful to minimize erases with flash-based indices
  - The **stepped-merge variant of LSM trees is used in many BigData storage systems**
    - Google BigTable, Apache Cassandra, MongoDB
    - And more recently in SQLite4, LevelDB, and MyRocks storage engine of MySQL

Buffer Tree

- Alternative to LSM tree
- Key idea: each internal node of B+-tree has a buffer to store inserts
  - Inserts are moved to lower levels when buffer is full
  - With a large buffer, many records are moved to lower level each time
  - Per record I/O decreases correspondingly
- Benefits
  - Less overhead on queries
  - Can be used with any tree index structure
  - Used in PostgreSQL Generalized Search Tree (GiST) indices
- Drawback: more random I/O than LSM tree



Bitmap indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- 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*
    - Particularly easy if records are of fixed size
- 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(Cont'd)

- 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
- Example

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

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- 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)
- 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

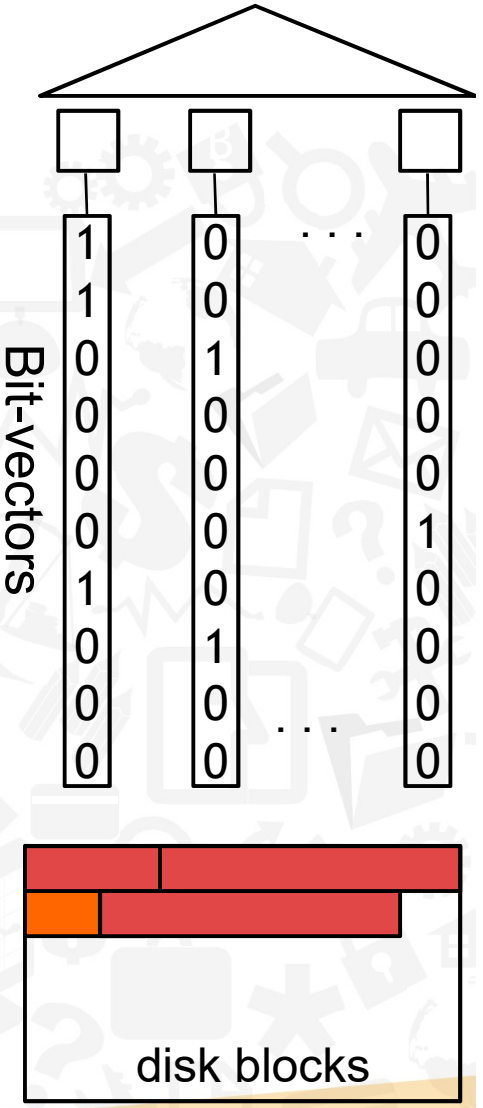
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g., if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
  - E.g., 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts

Bitmap Indices (Cont.)

- 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
  - Above technique merges benefits of bitmap and B$^+$-tree indices
- Also, bitmaps can be efficiently compressed in a ***run-length*** encoding scheme



Bitmap Compression

0001000000001000000000000000011 00000000000 . . . 00000000000000 00000000000000000000000000000001 00

(a) An original bit-vector with 8,000 bits

31 bits	256* 31 bits	31 bits	2 bits

(b) Grouping as a unit of 31 bits and Merging identical groups

000010…010…011	**10**0… 0100000000	**0**00…001	000…0**00**
31 literal bits	Run-length is 256		Remaining word
Uncompressed word	Compressed word		

(c) Encoding each group as 1 word (4byte on a 32-bit machine)

Cursor ***C***

={ position, //Integer position value (Logical address)
word, // The current word C is located at.
bit, // The position of the bit C is visiting, in C.word
rest } //The bit position in the remaining word

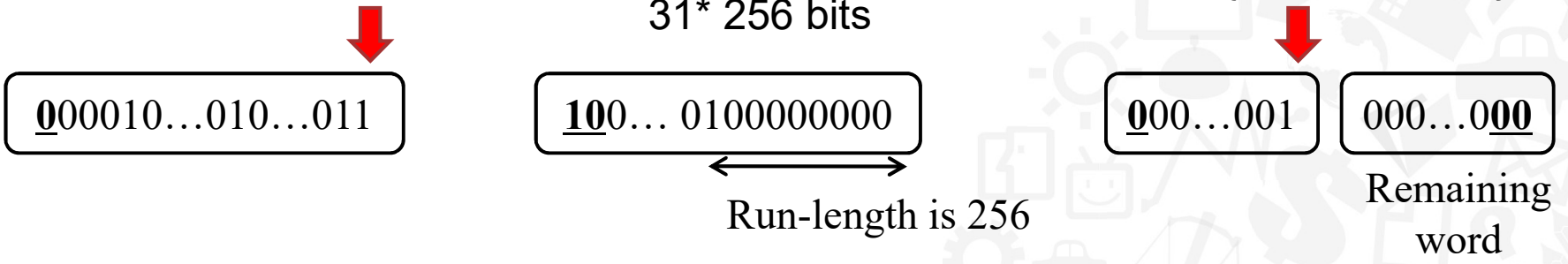
Computation with Compressed Bitmaps

a) Get the position of the next 1

C = {31, 0, 31,0}

Skip to examine
31* 256 bits

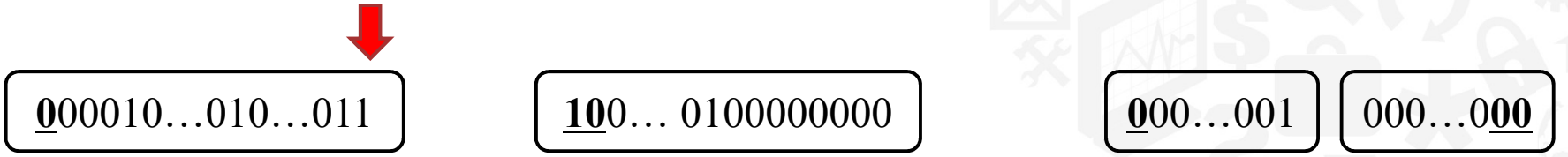
C={7998, 2, 31, 0}



b) Check a bit value at the position 3,000

C = {31, 0, 31,0}
with distance to move,
2,869=(3000-31)

Since 31* 256 > 2,869,
The bit we find is within the word 1.

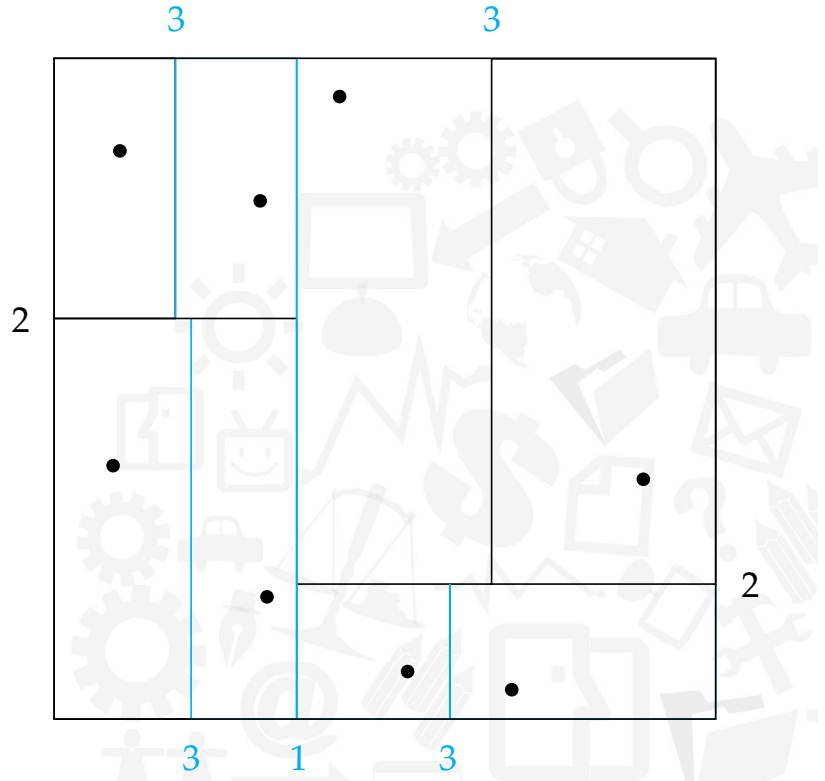


Spatial Data

- Databases can also store data types such as lines, polygons, in addition to raster images
  - allows relational databases to store and retrieve spatial information
  - Queries can use spatial conditions (e.g., contains or overlaps)
  - queries can mix spatial and nonspatial conditions
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions
- **Range queries** deal with spatial regions
  - e.g., ask for objects that lie partially or fully inside a specified region
- Queries that compute intersections or unions of regions
- **Spatial join** of two spatial relations with the location playing the role of join attribute

Indexing Spatial Data

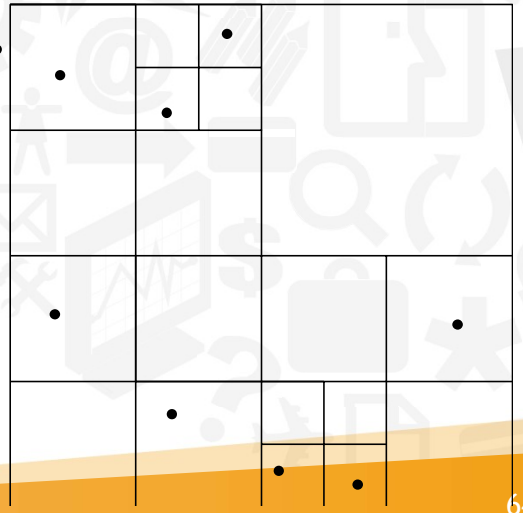
- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - Choose one dimension for partitioning at the root level of the tree.
  - Choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.



The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).

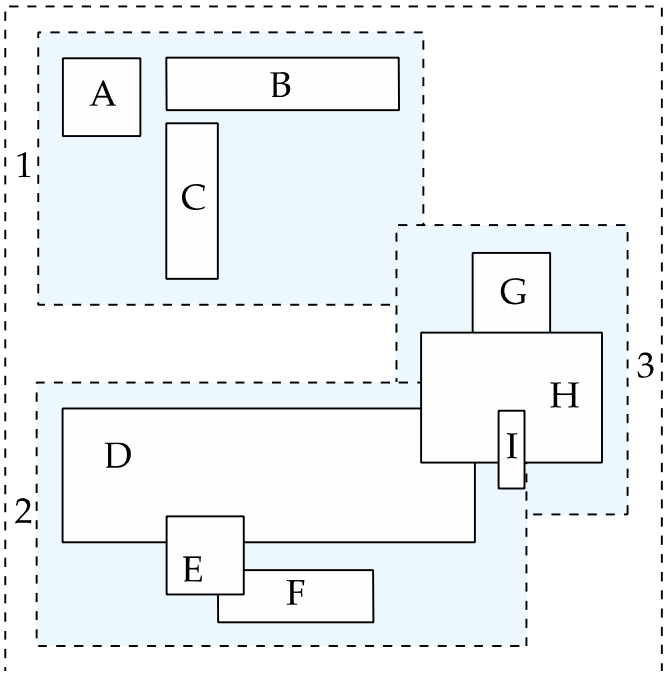


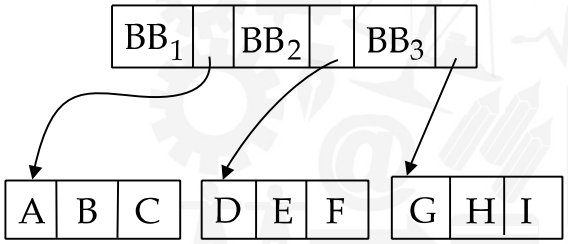
R-Trees

- **R-trees** are a N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R^+ -trees and R*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*

Example of R-Tree

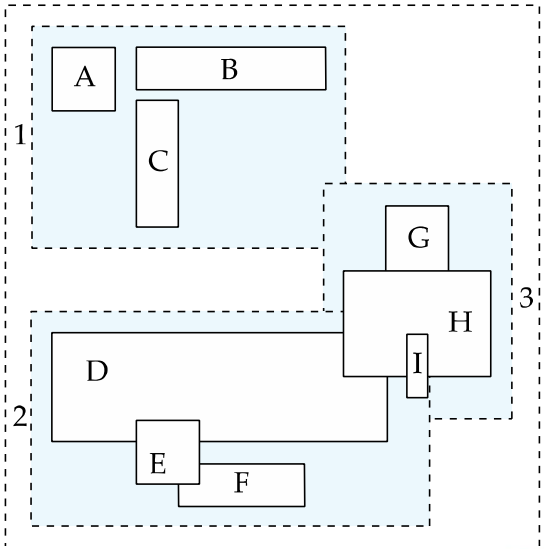
- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.

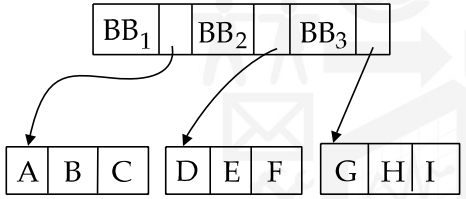




Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched, but works acceptably in practice.





Indexing Temporal Data

- Temporal data refers to data that has an associated time period (interval)
  - Example: a temporal version of the *course* relation

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

- Time interval has a start and end time
  - End time set to infinity (or large date such as 9999-12-31) if a tuple is currently valid and its validity end time is not currently known
- Query may ask for all tuples that are valid at a point in time or during a time interval
  - Index on valid time period speeds up this task

Indexing Temporal Data (Cont.)

- To create a temporal index on attribute *a*:
  - Use spatial index, such as R-tree, with attribute *a* as one dimension, and time as another dimension
    - Valid time forms an interval in the time dimension
  - Tuples that are currently valid cause problems, since value is infinite or very large
    - Solution: store all current tuples (with end time as infinity) in a separate index, indexed on (*a, start-time*)
      - To find tuples valid at a point in time *t* in the current tuple index, search for tuples in the range (*a, 0*) to (*a,t*)
- Temporal index on primary key can help enforce temporal primary key constraint

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

Question?

–source: https://www.fox.com/the-simpsons