数据库系统原理

Database System Principle

邵蓥侠

Email：shaoyx@bupt.edu.cn

北京邮电大学计算机学院

计算机应用技术中心

PART 5

DATA STORAGE AND QUERY

Chapter 14

Indexing and Hashing

- Basic Concepts
- Ordered Indices

（ 略

- B$^+$-Tree Index Files
- B-Tree Index Files

）

- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

Contents in This Chapter

- Basic concepts and classification of **indexing**
  - ordered indices, *hash indices*
- Properties/types of ***ordered*** indices
  - primary/clustering indices, secondary/non-clustering indices
  - dense indices, sparse indices
  - single-level indices, multi-level indices (e.g. B^+-tree, B-tree)
- ***Hash*** indices
  - hash functions
  - static hash, dynamic hash

目 标！！！

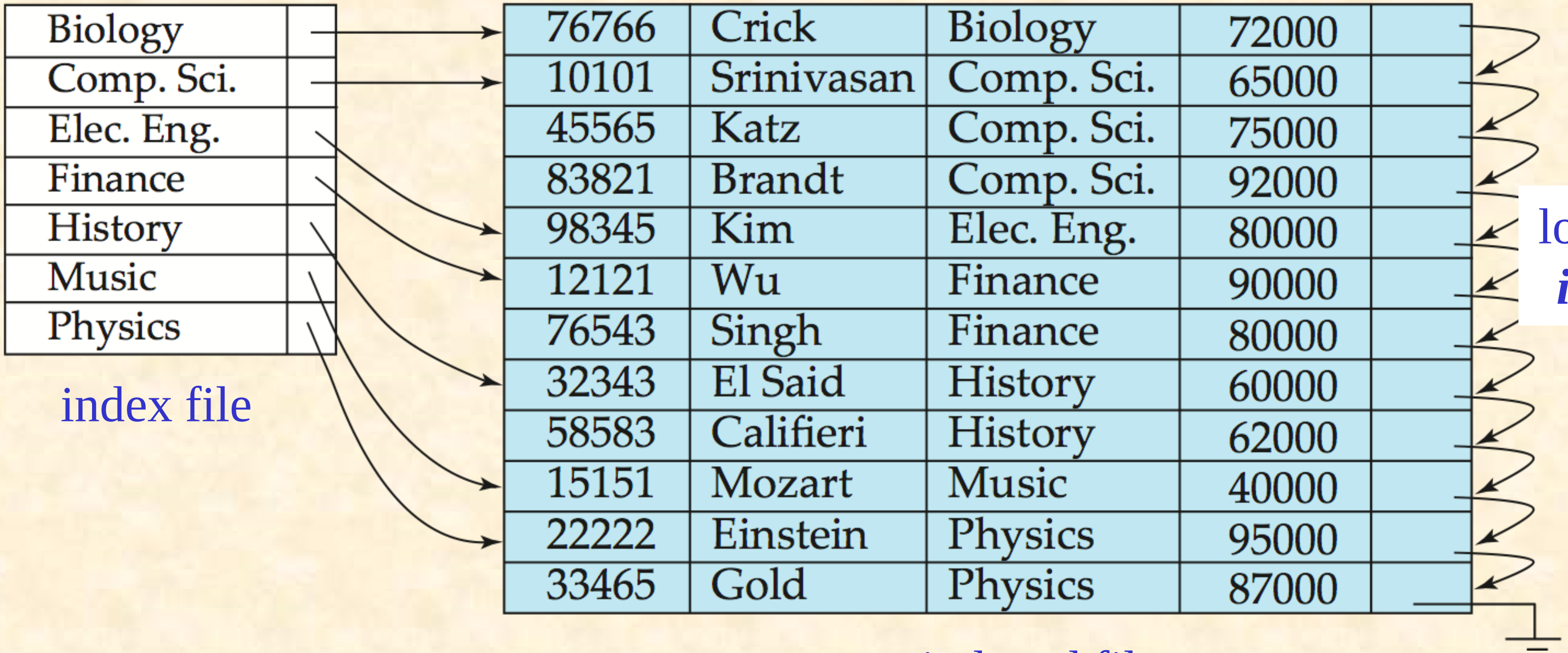
- 学会正确建立、管理索引，提高数据访问速度
  - 索引类型
  - **select**、**insert**、**update**时的索引管理
- 了解索引对**select/update/insert/delete**的影响
- 实验6. 数据库物理设计
  实验7. 查询优化

§ 14.1 Basic Concepts

- How to locate tuples/records in DB file quickly?
- **Indexing** (索引技术) mechanisms
  - used to speed up access to desired data
  - e.g. for relation instructor(*ID*, *name*, *dept_name*, *salary*)
  - shown in Fig.14.1, the index

    dept_name → physical address of record (i.e.tuple) in DB file ***instructor***
- **Search Key**
  - attributes or a set of attributes used to look up the records in a file
  - e.g. *dept_name*

instructor(*ID*, *name*, *dept_name*, *salary*)



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

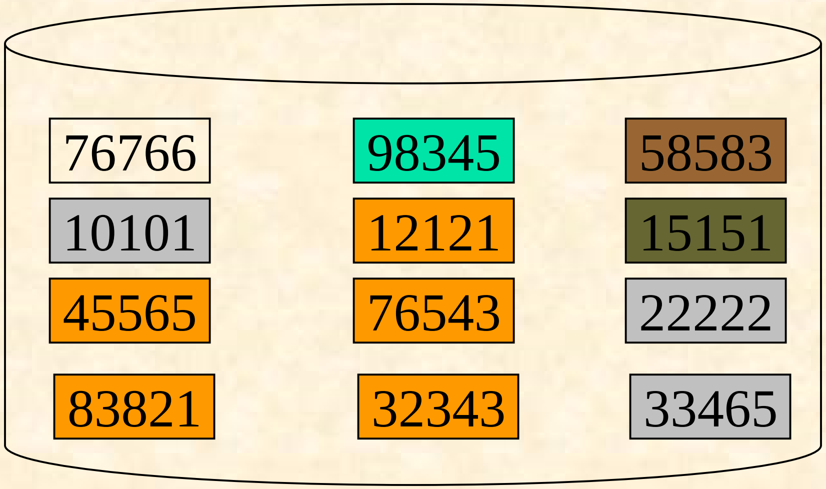
76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein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

index file

logical file ***instructor***

indexed file

Note: the file ***instructor*** is logically a sequential file, but its records may be stored *non-contiguously or non-ordered* on the disk



76766	98345	58583
10101	12121	15151
45565	76543	22222
83821	32343	33465

physical file ***instructor***

Fig. 14.1 DB *indexed* file **instructor** and its *index* file

Basic Concepts (cont.)

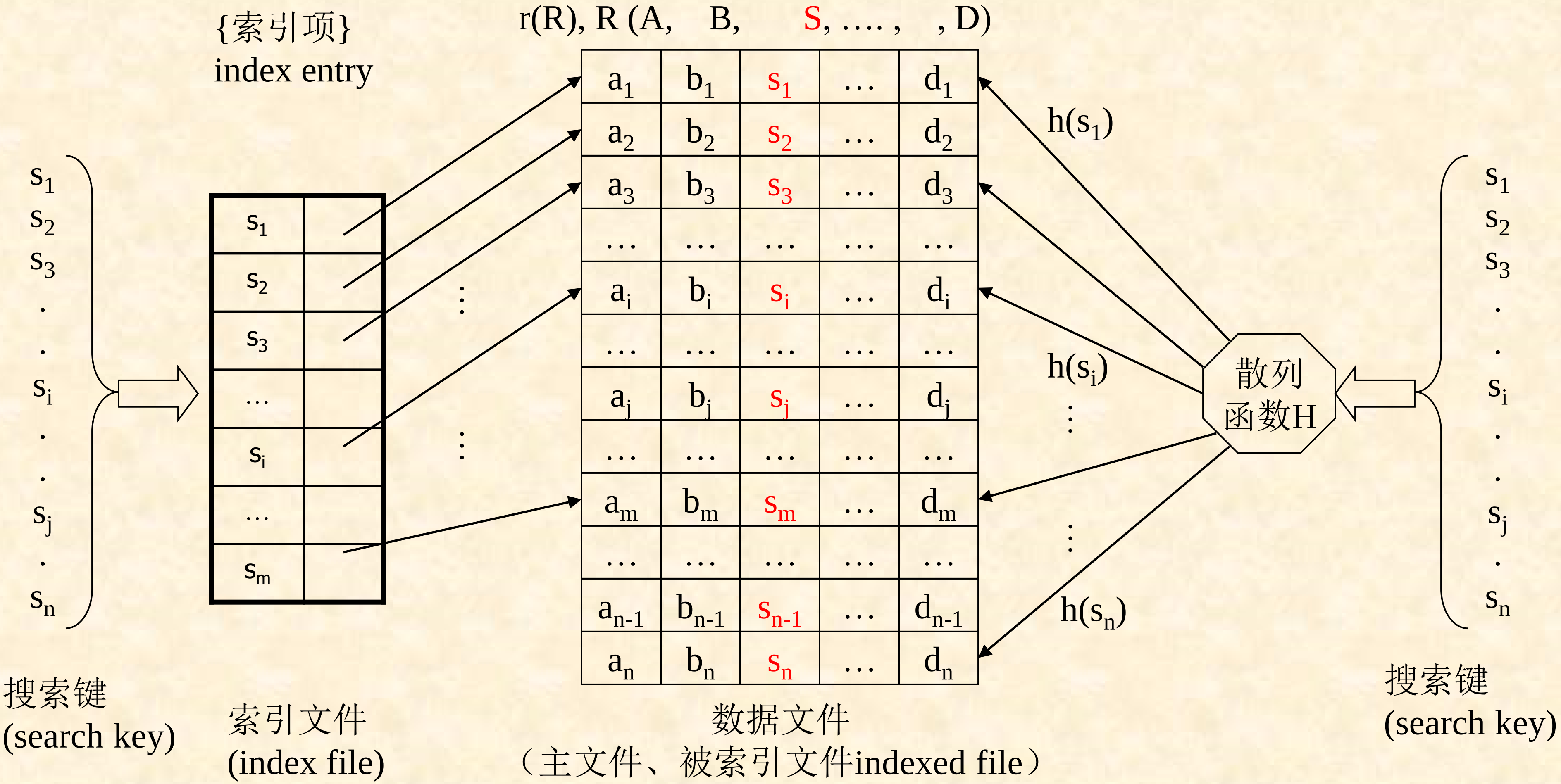
- An index file consists of records (called index entries) of the form
- Index files are typically much smaller than the original file

search-key	pointer

- Indexing
  - mapping from *search-key* to storage *locations* of the file records,

    i.e. search-key → storage locations of the records in disks

ordered indices **hash indices**


{索引项}
index entry
s_1 s_2 s_3 . . s_i . . s_j . s_n
搜索键
(search key)
索引文件
(index file)
r(R), R (A, B, S, …. , , D)
数据文件
（主文件、被索引文件indexed file）
h(s_1)
h(s_i)
h(s_n)
散列
函数H
s_1 s_2 s_3 . . s_i . . s_j . s_n
搜索键
(search key)

s_1	
s_2	
s_3	
…	
s_i	
…	
s_m	

A	B	S	….	D
a_1	b_1	s_1	…	d_1
a_2	b_2	s_2	…	d_2
a_3	b_3	s_3	…	d_3
…	…	…	…	…
a_i	b_i	s_i	…	d_i
…	…	…	…	…
a_j	b_j	s_j	…	d_j
…	…	…	…	…
a_m	b_m	s_m	…	d_m
…	…	…	…	…
a_{n-1}	b_{n-1}	s_{n-1}	…	d_{n-1}
a_n	b_n	s_n	…	d_n

图 14.0.1 索引技术(indexing)及其分类

Basic Concepts (cont.)

- Two basic kinds of indices:
  - **ordered indices:** the index file is used to store the *index entries* in which *the search key* of the records and the *address* of the records are stored in sorted order
  - **hash indices:** the "*hash function*" is used to map the *the search key* of the records to the *address* of the records
  - the records are stored in the "buckets", the *number* of the bucket is as the address of the records and is determined by the *hash function*

Basic Concepts (cont.)

- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead（空间开销）

§ 14.2 Ordered Indices

- DBS files with indexing mechanism include two parts :
  - ***indexed*** file, in which data **records** are stored
  - ***index file***, in which **index entries** are included
  - e.g. Fig.14.1 ▷
- The indexed file can be organized as
  - sequential file
  - heap file
  - hash file
  - clustering file

Ordered Indices (cont.)

- **Ordered index** (in index file)
    - the *index entries* in the index file are stored in some sorted order, for instance, in accordance with the order of the search key

e.g. in Fig.14.1(   ) , the index file is sorted by ***dept_name***

▷

§ 14.2.1 Primary and Secondary Indices

- **Primary/clustering index** (in the index file)
  - considering a ***index file*** and its corresponding ***indexed file***.
  - the ***indexed file*** is a ***sequential file***, and the index whose search key also specifies the sequential order of the ***indexed file***
  - /* 索引文件的搜索键所规定的顺序与被索引的顺序文件中的纪录顺序一致
  - e.g. in Fig.14.1▷ , (*dept_name*, address of records) defines the same orders of the records as that in sequential indexed file ***instructor***
  - note: also called **clustering index**

Primary and Secondary Indices (cont.)

- The search key of a primary index is usually but not necessarily the primary key
  - e.g. in Fig. 14.1, ***dept-name*** is not the primary key of ***instructor***

Primary and Secondary Indices (cont.)

- 聚集索引 vs 主索引

  在实际数据库系统中，如SQL Server、DB2等，

  - 聚集索引：

    索引文件的搜索键所规定的顺序与被索引的顺序文件中的纪录顺序一致

  - 主索引：

    建立在主键（主属性上）的索引

  - 当一个表定义了主键后，DBMS会自动为该表在主键上建立聚集索引，该索引同时又是主索引

Primary and Secondary Indices (cont.)

- 一个表上只能建立一个聚集索引，也只能有一个主索引。— why？ 数据文件根据索引项进行排序，只能有一种排列顺序
- 但可以建立多个非聚集索引
- 如果一个表上没有定义主键，则不会有主索引；但可以为该表建立一个聚集索引
- **主索引一定是聚集索引，但聚集索引不一定是主索引**

Primary and Secondary Indices (cont.)

- 实验：对比分析
  - 在没有定义主键的关系表中插入数据：

    在关系表所在数据库文件中，各个元组/行/记录顺序一般是无序，取决于数据插入顺序——heap文件

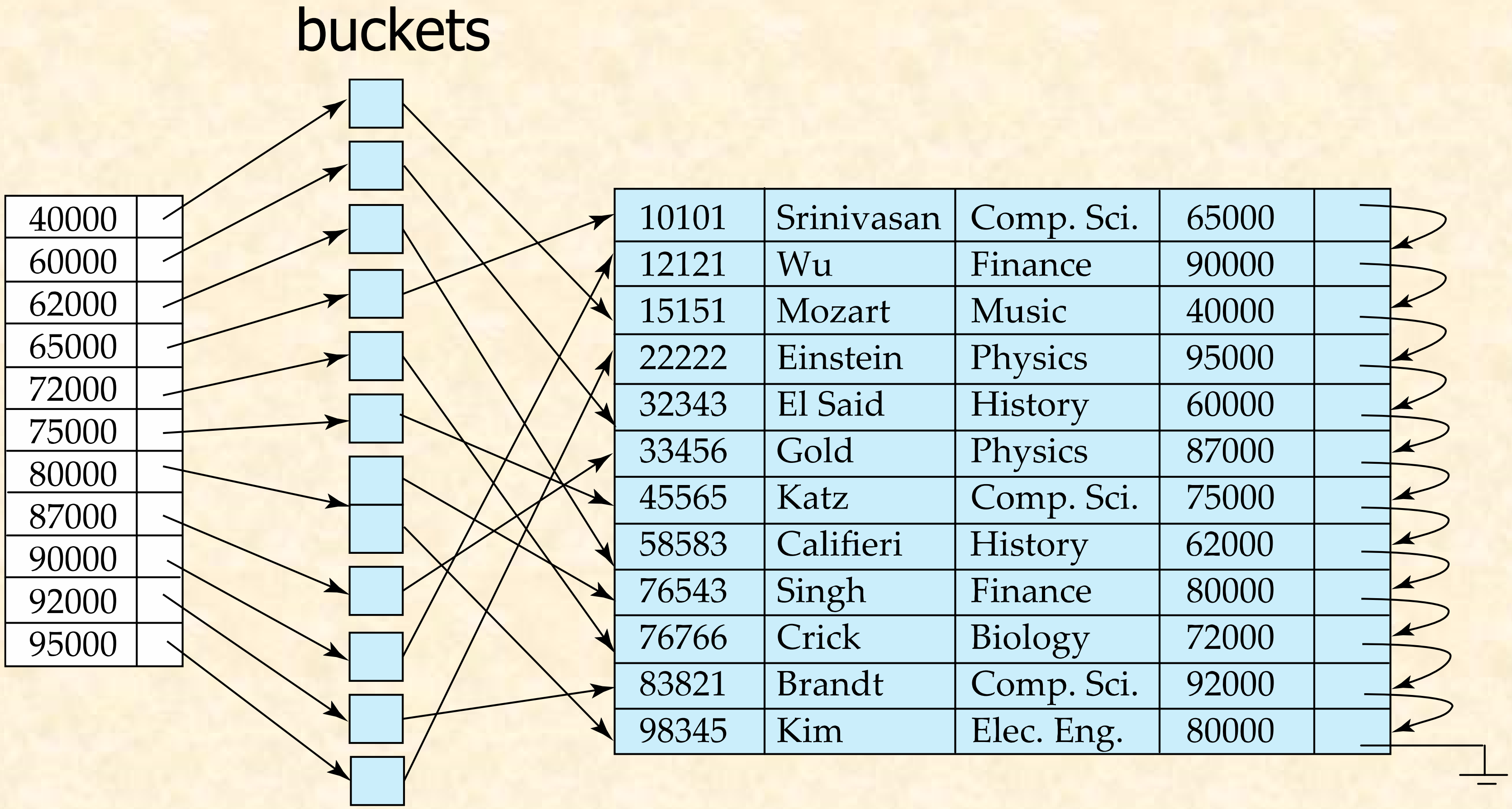
e.g. select * from *tablename*

  - 在有主键的关系表中插入数据：

    数据库文件是有序的，按照主键顺序排列

14.2.4 Primary and Secondary Indices (cont.)

- **Secondary index**
  - an index whose search key specifies an order different from the sequential order of the file.
  - also called **non-clustering index**
- Secondary indices have to be *dense indices*
- Index-sequential file (索引顺序文件)
  - ordered sequential file with a primary/clustering index on the search key
  - e.g. Fig.14.1 ▷



40000
60000
62000
65000
72000
75000
80000
87000
90000
92000
95000

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

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

14.2.1 Dense and Sparse Indices

- **Dense index**
  - the index record in the index file appears for every search-key value in the indexed file（文件中的每个搜索码值有一个索引记录）
  - each value of search-key in *the indexed file* corresponds to an index entry in *the index file*
  - e.g. Fig.14.1 ▷

dense index on *dept_name*, with *instructor* file sorted on *dept_name*

Dense and Sparse Indices

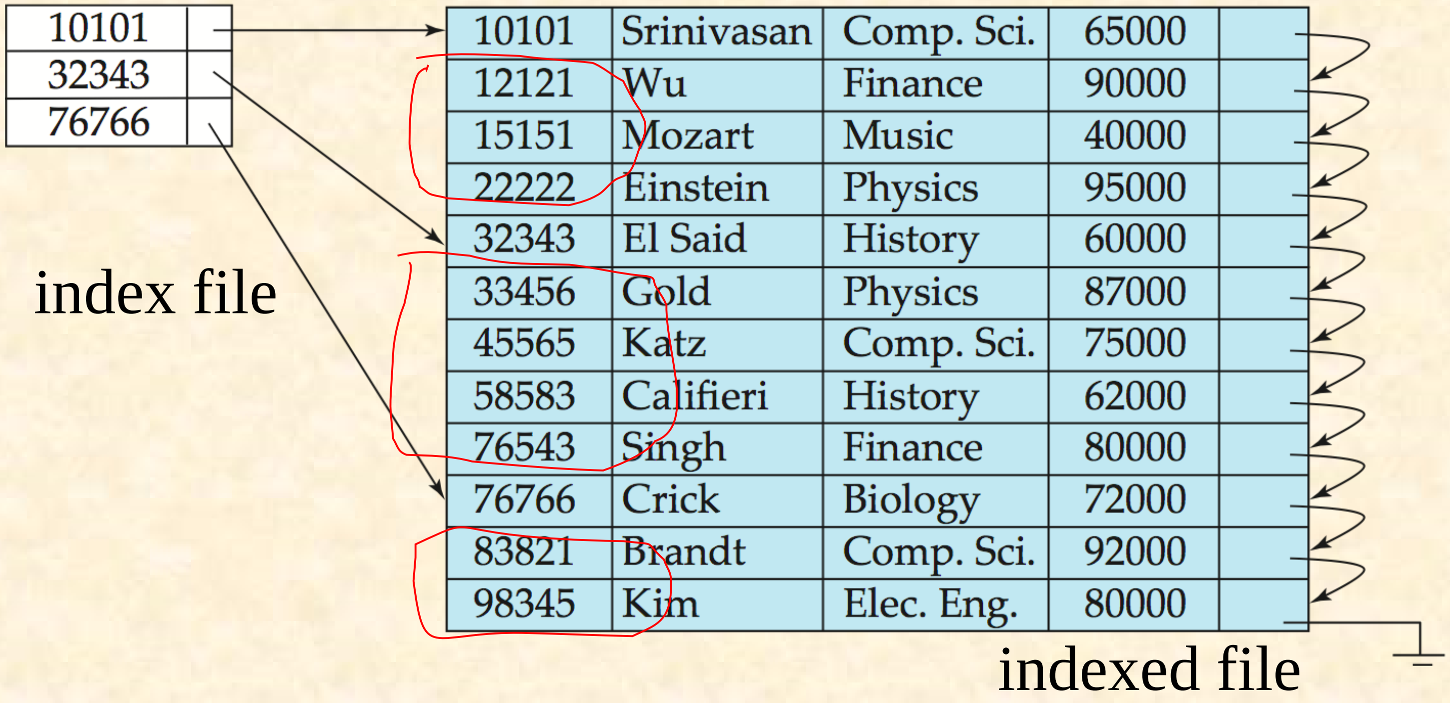
- In a dense primary index, the index record contains the search-key value and a pointer to the ***first*** record with that search-key value
  - the rest of the records with the same search-key value would be stored sequentially after the first record
  - e.g. Fig.14.1,▷ search-key value "*Comp.Sci*" corresponds to three records in the file

Dense and Sparse Indices (cont.)

- **Sparse Index**
  - index file contains index entries for only some search-key values in the indexed file（只为搜索码的某些值建立索引记录）
  - e.g. Fig.14.3 ▷

- To locate a file record with search-key value *K*
  - find index entry with largest search-key value ⩽ *K*
  - search file sequentially starting at the record to which this index entry points

Dense and Sparse Indices



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	

Fig.14.3 Sparse index for the file ***instructor***

Primary vs Secondary Indices

- Indices offer substantial benefits when searching for records.
- **BUT:**

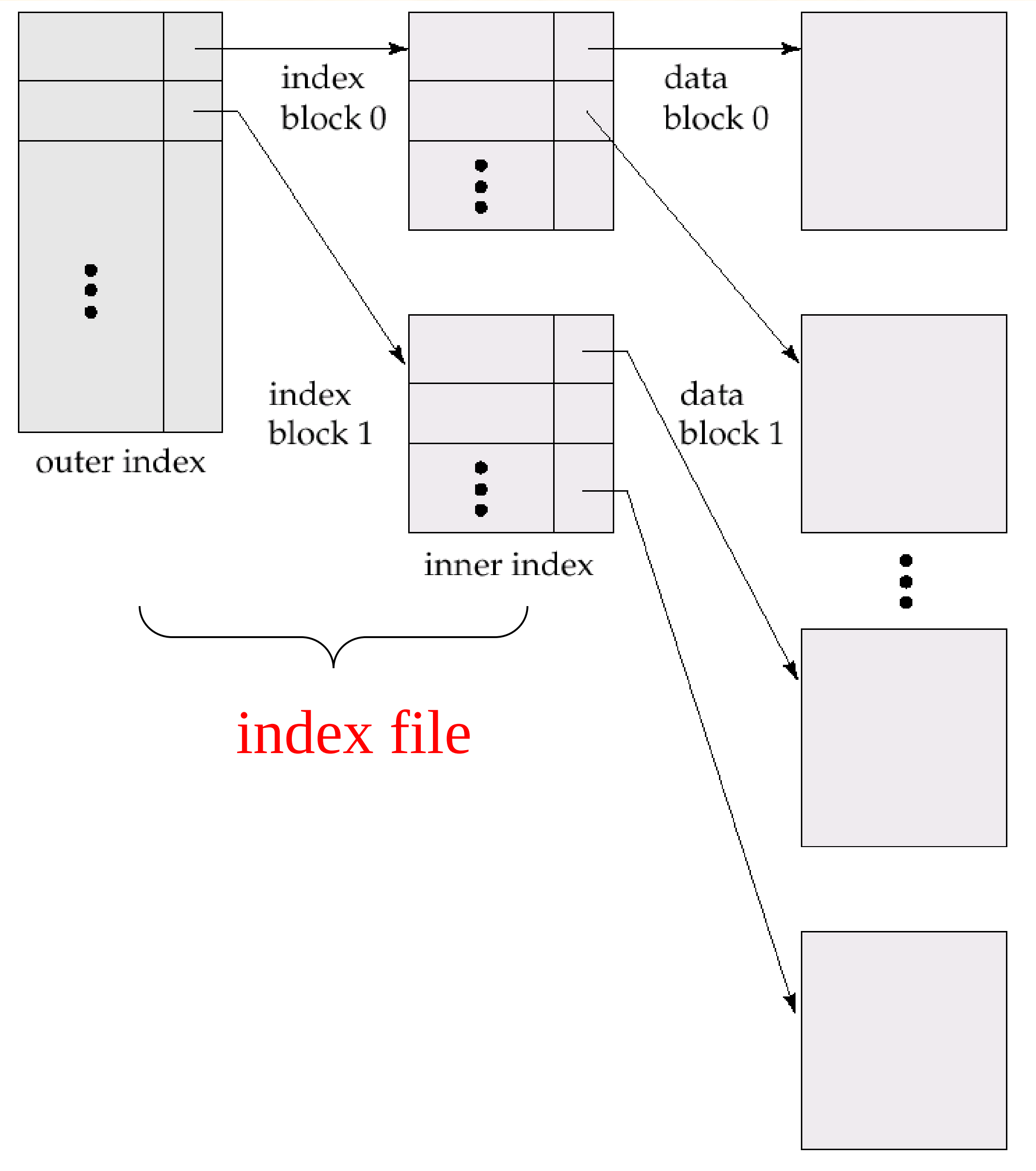
  Updating indices imposes overhead on database modification -- when a file is modified, every index on the file must be updated

- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds（毫秒）, versus about 100 nanoseconds（纳秒） for memory access

14.2.2 Multi-level Indices

- The index file may be very large, and cannot be entirely kept in memory
- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index file
  - inner index – the primary 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.

B^+-tree indices and B-tree indices are two types of ***efficient*** multi-level indices, and widely used in DBS and MS file systems



index file

indexed file

Fig. 14.5
Two-level sparse index

Note: similar to hierachical page tables or file indexs in OS

14.6.2 Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*dept_name*, *salary*)
- E.g.

  create index *Mutiple-index* on *takes(course_id, semester, year)*

- Lexicographic ordering （字典序）: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$, or
  - $a_1=b_1$ and $a_2 < b_2$

左前缀原则

Suppose we have an index on combined search-key

(*dept_name*, *salary*).

- With the **where** clause

  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

- Can also efficiently handle

  where *dept_name* = “Finance”

左前缀原则

- But cannot efficiently handle

1.

where *dept_name* < “Finance” **and** *balance* = 80000

  - may fetch many records that satisfy the first but not the second condition
  - **the index is not used efficiently**

2.

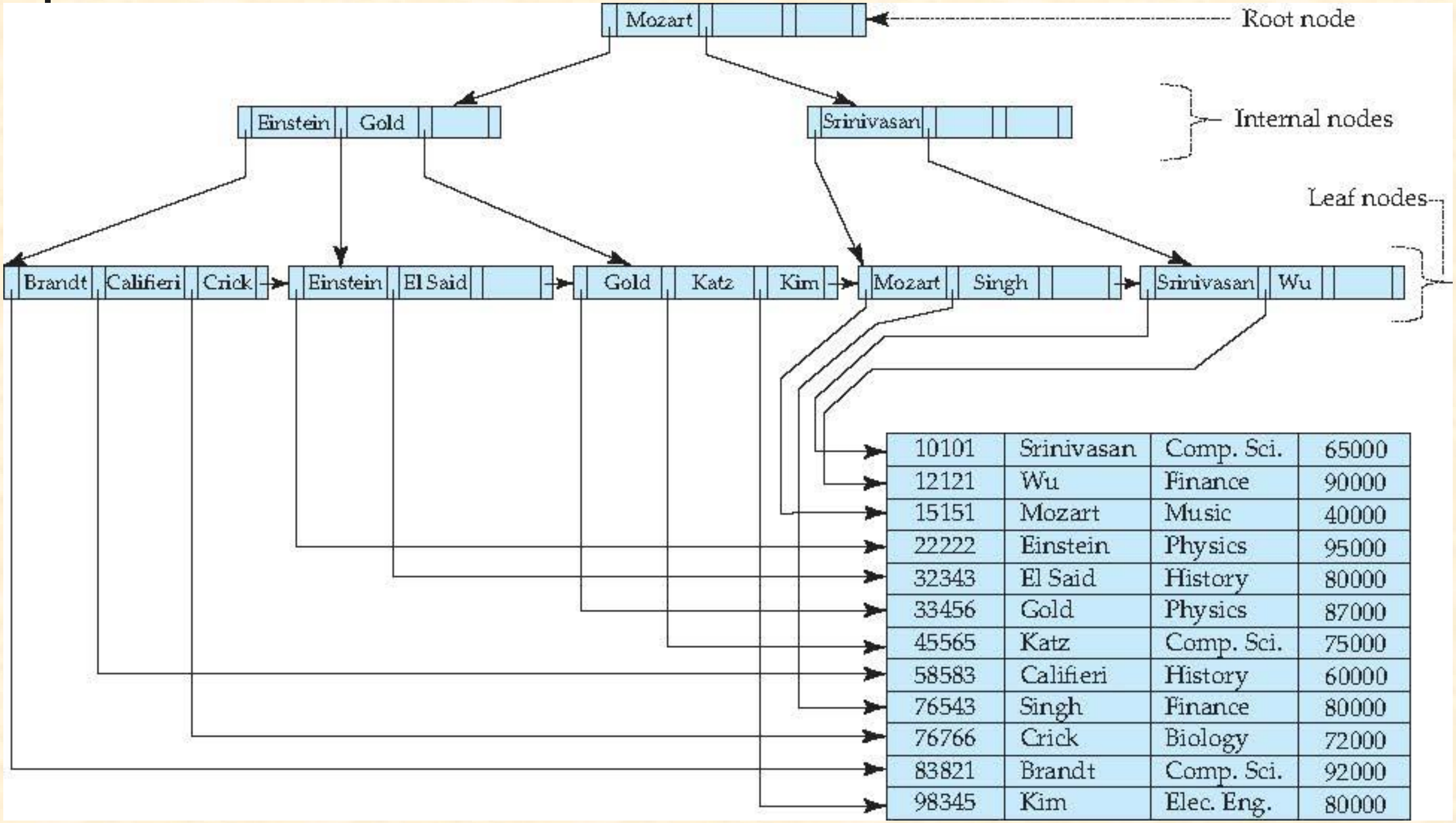
where *balance* = 80000

  - **the index is not used for query**

14.3/14.4 B$^+$-Tree/B-Tree Index Files

- B$^+$-tree indices are an alternative to indexed-sequential file
- Extensively used multi-level index, with advantages
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - reorganization of entire file is not required to maintain performance.

Example of B$^+$-Tree



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 (Cont.)

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

- All paths from root to leaf 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.

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).
- The search-keys in a node are ordered

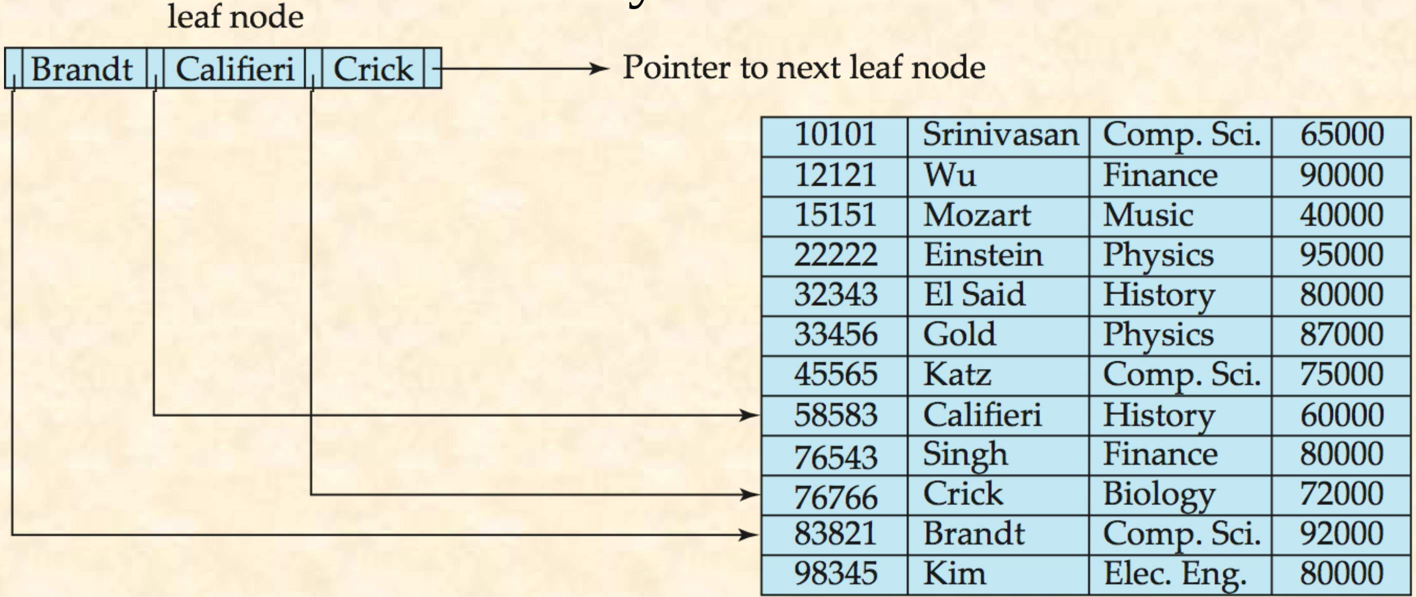
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B^+-Trees

Properties of a leaf node:

- 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



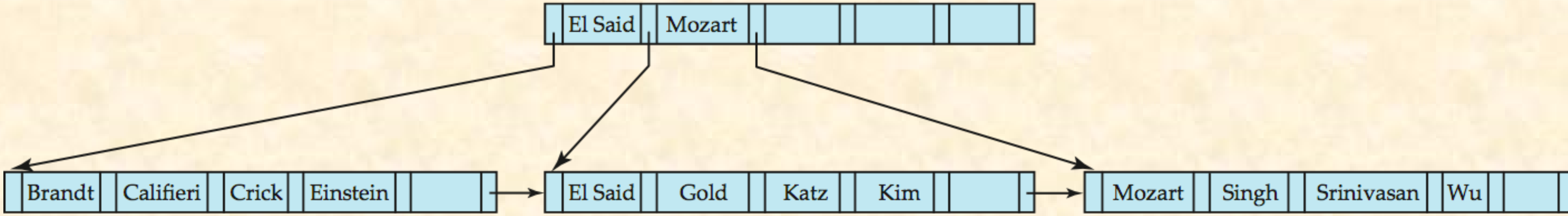
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
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 $2 \le i \le n - 1$, 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}

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

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.

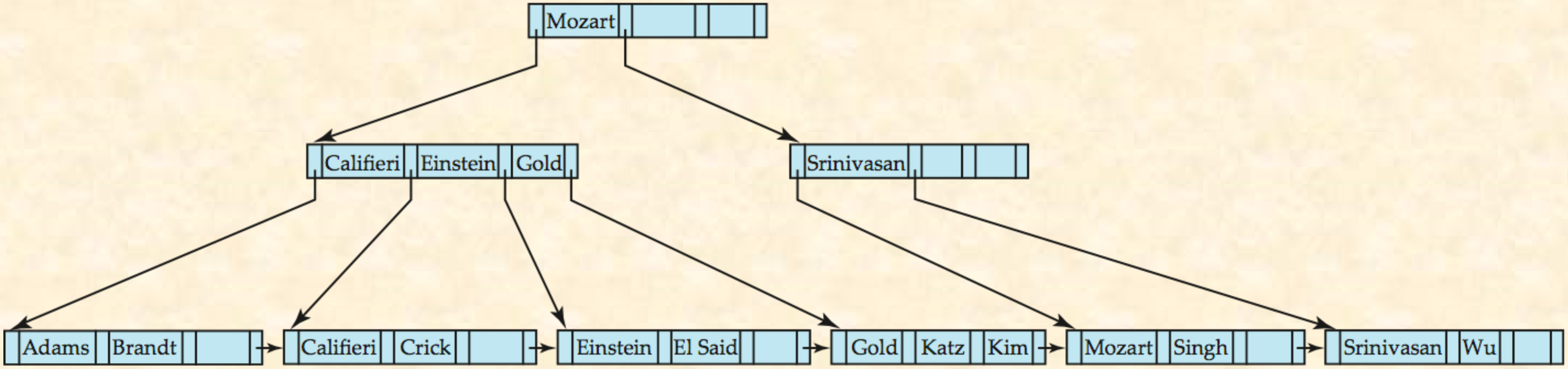
B^+-trees Performance

- 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 (as we shall see).

Queries on B^+-Trees

- Find record with search-key value *V*.

```
1. $C = root$
2. While C is not a leaf node {
   1. Let $i$ be least value s.t. $V \leq K_i$.
   2. If no such exists, set $C$ = last non-null pointer in $C$
   3. Else { if (V= $K_i$ ) Set C = $P_{i+1}$ else set $C = P_i$}
   }
3. Let $i$ be least value s.t. $K_i = V$
4. If there is such a value $i$, follow pointer $P_i$ to the desired
   record.
5. Else no record with search-key value $k$ exists.
```



Queries on B^{+-}Trees (Cont.)

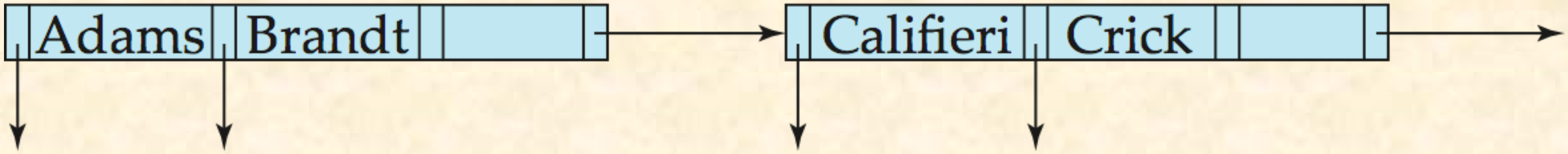
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- 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

Insertion on B^{+}-Trees

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
   1. Add record to the file
   2. If necessary add a pointer to the bucket.
3. If the search-key value is not present, then
   1. add the record to the main file (and create a bucket if necessary)
   2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
   3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

Insertion (Cont.)

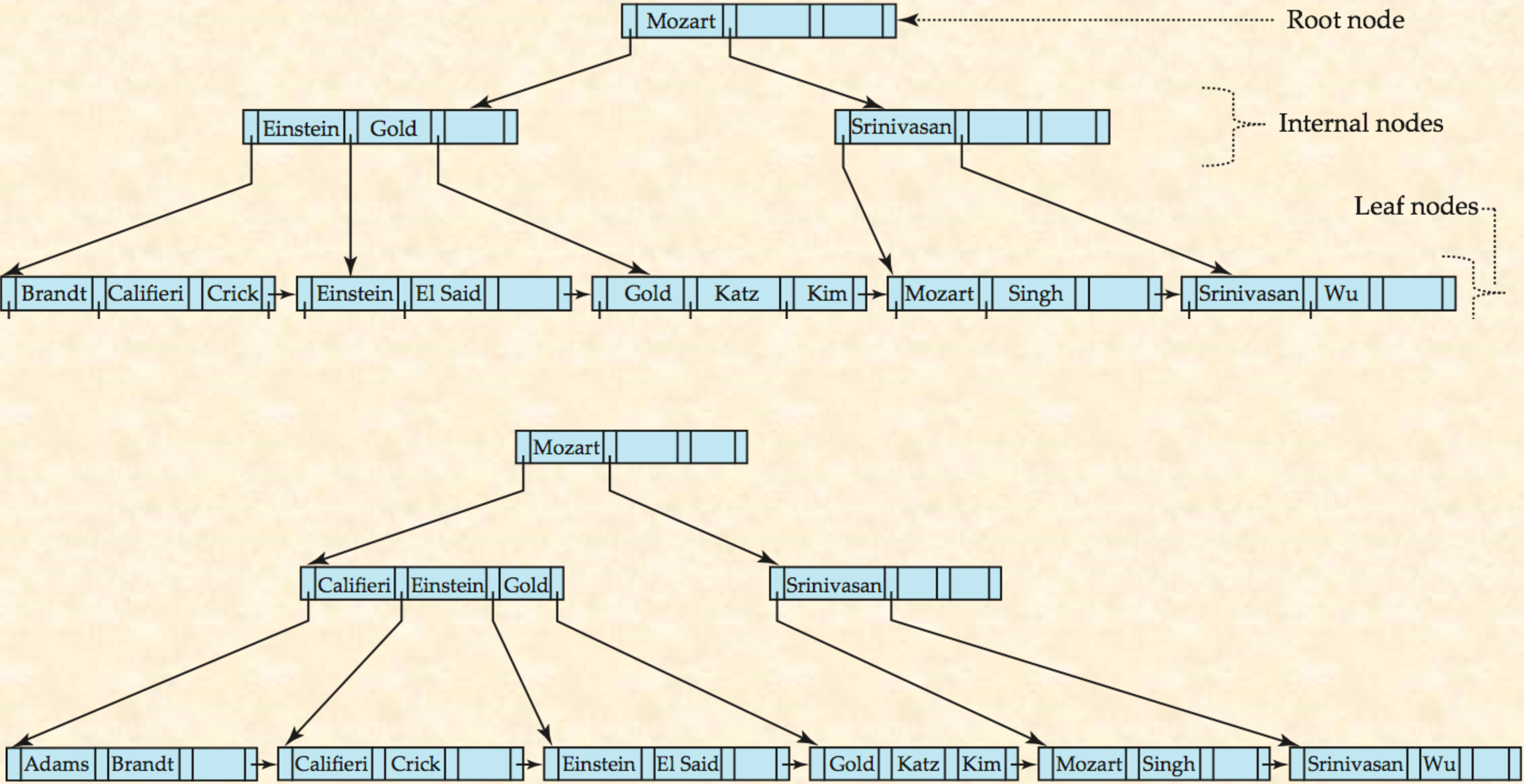
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first ⌈*n*/2⌉ in the original node, and the rest in a new node.
  - let the new node be *p*, and let *k* be the least key value in *p*. Insert (*k*,*p*) in the parent 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

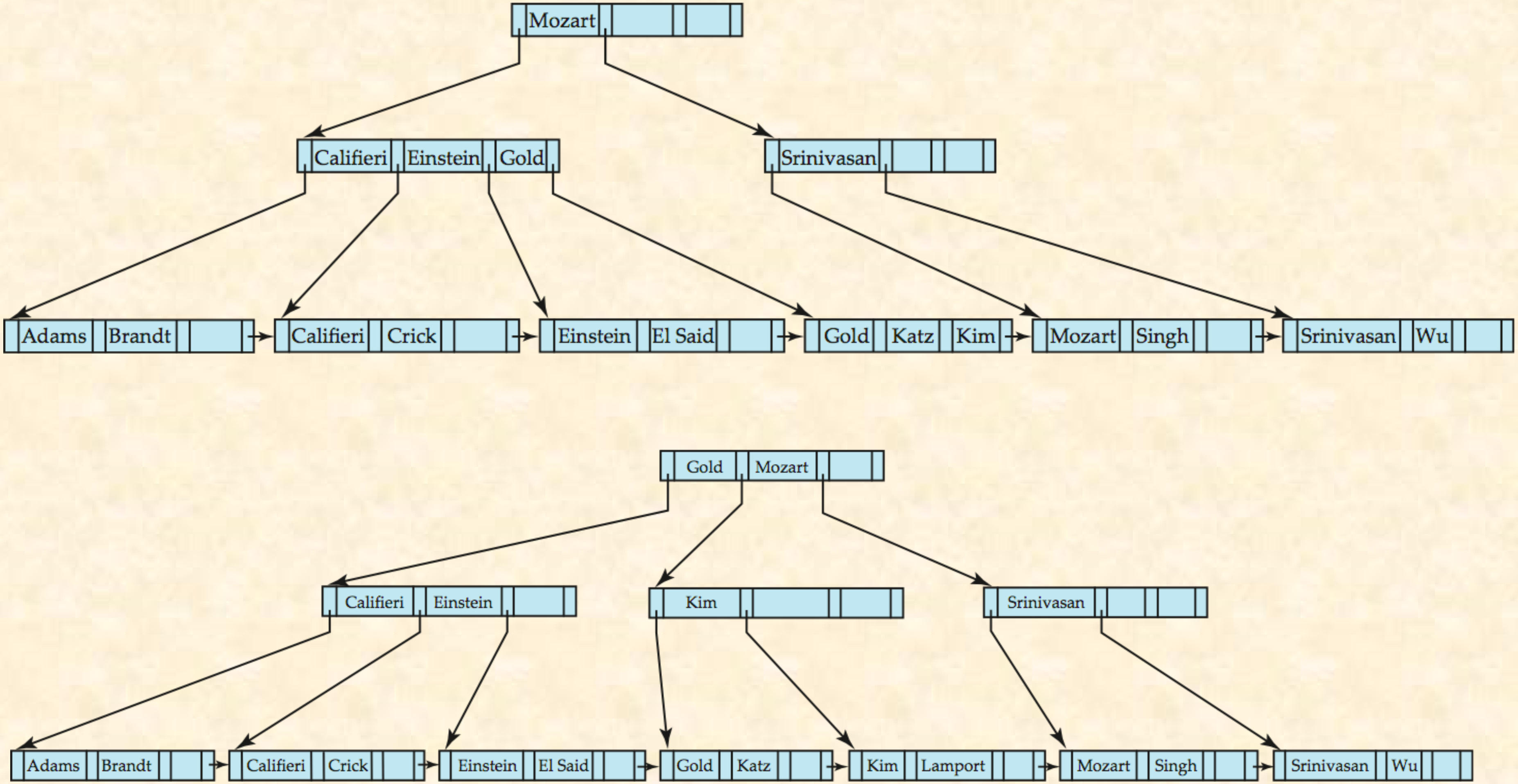
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

B+-Tree Insertion



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

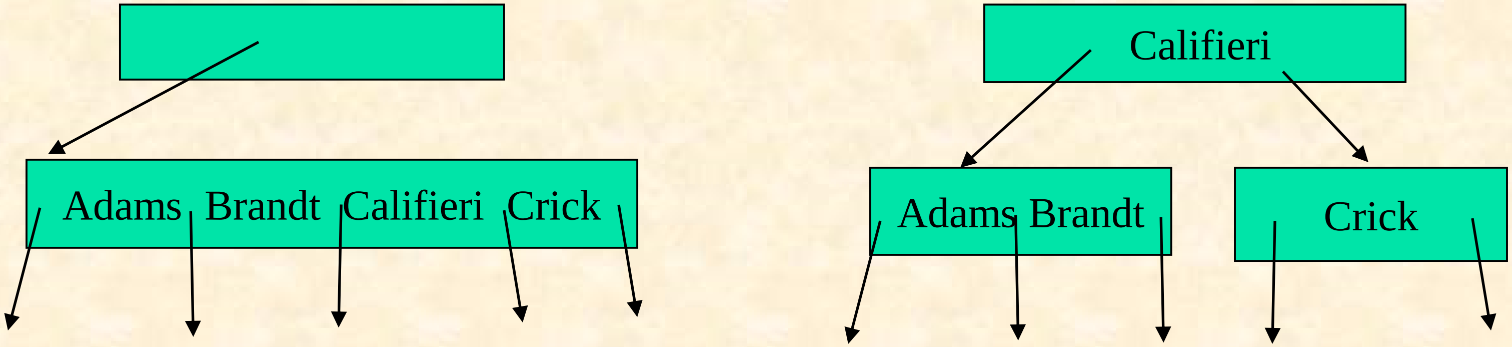
B^+-Tree Insertion



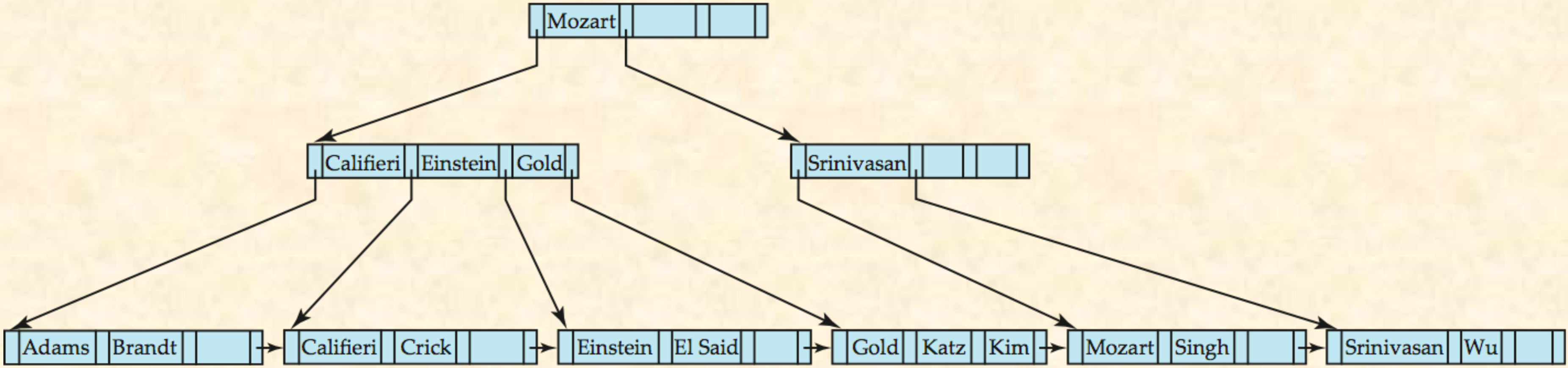
B^+-Tree before and after insertion of “Lamport”

Insertion in B^+-Trees

- 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



Examples of B$^+$-Tree Deletion



Mozart

Califieri Einstein Gold

Srinivasan

Adams Brandt

Califieri Crick

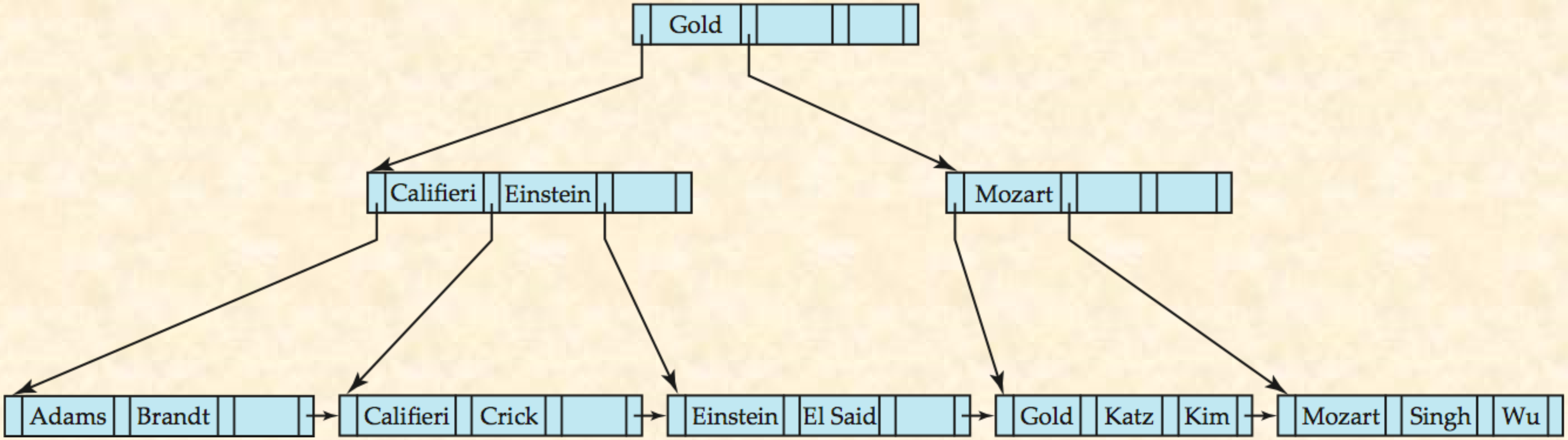
Einstein El Said

Gold Katz Kim

Mozart Singh

Srinivasan Wu

Before and after deleting "Srinivasan"



Gold

Califieri Einstein

Mozart

Adams Brandt

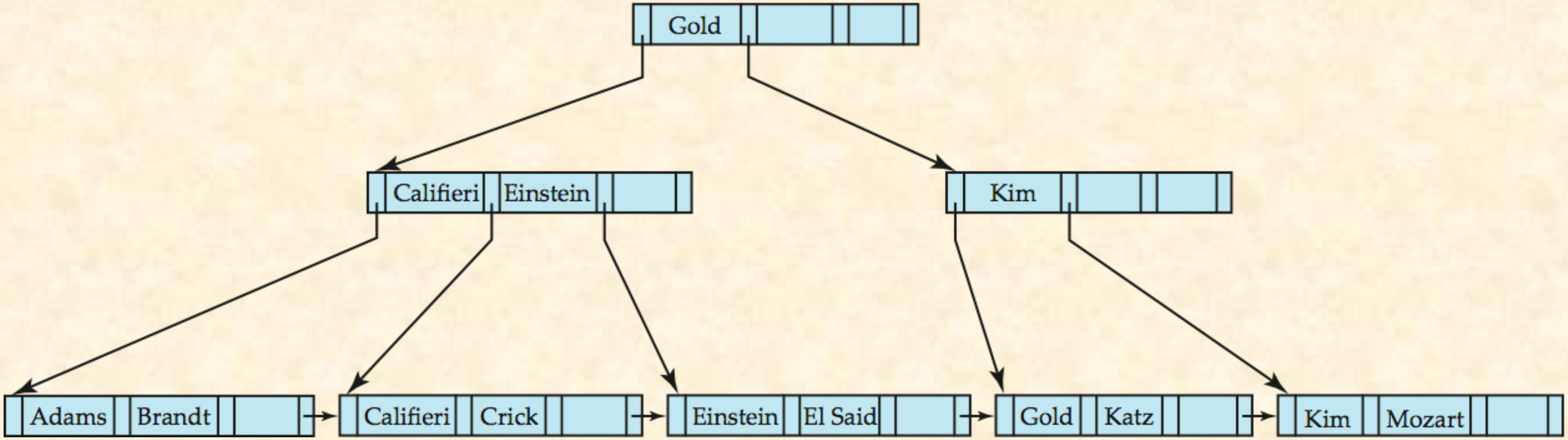
Califieri Crick

Einstein El Said

Gold Katz Kim

Mozart Singh Wu

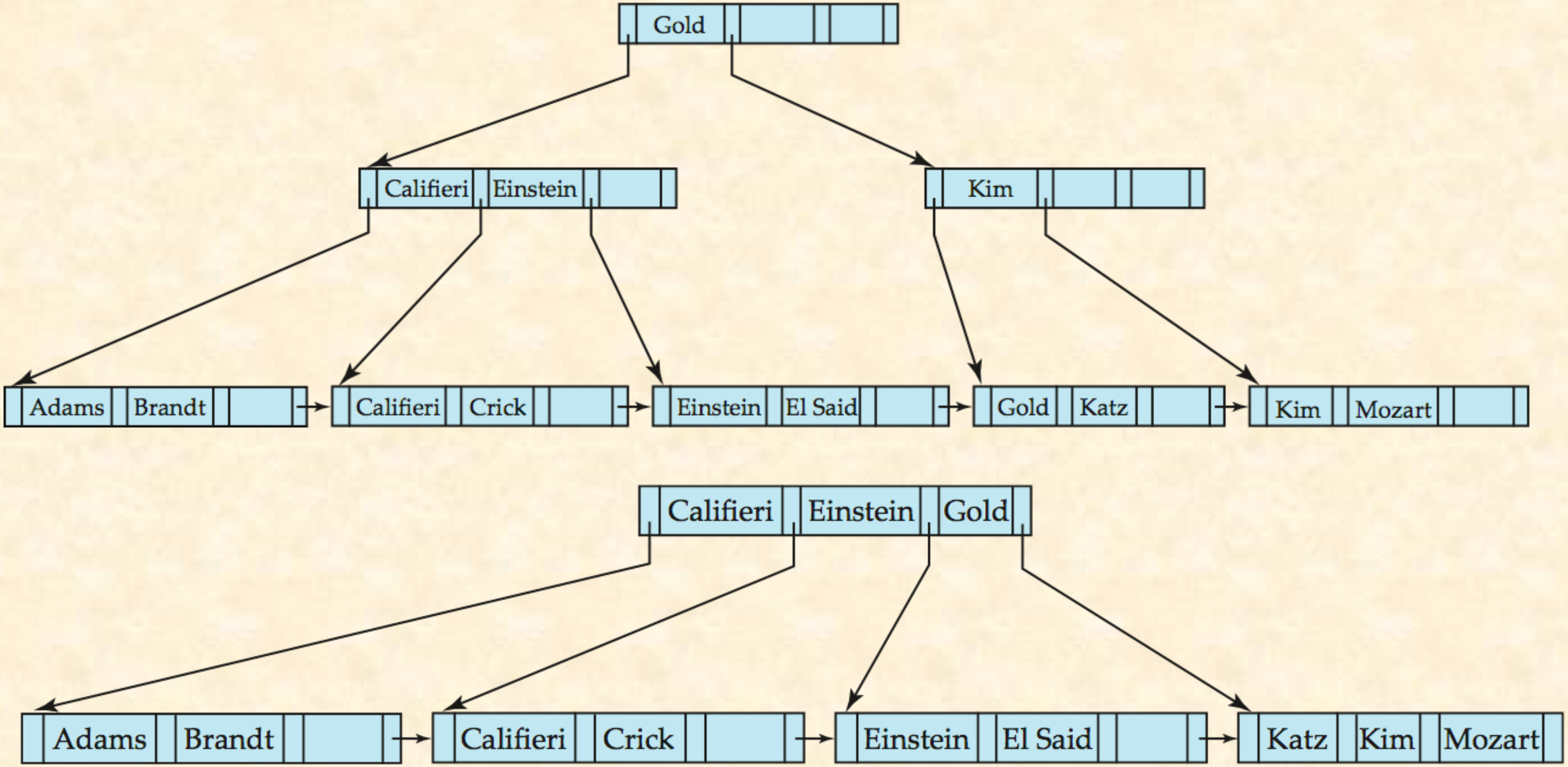
- Deleting "Srinivasan" causes merging of under-full leaves



Deletion of "Singh" and "Wu" from result of previous example

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



Before and after deletion of “Gold” from earlier example

- 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

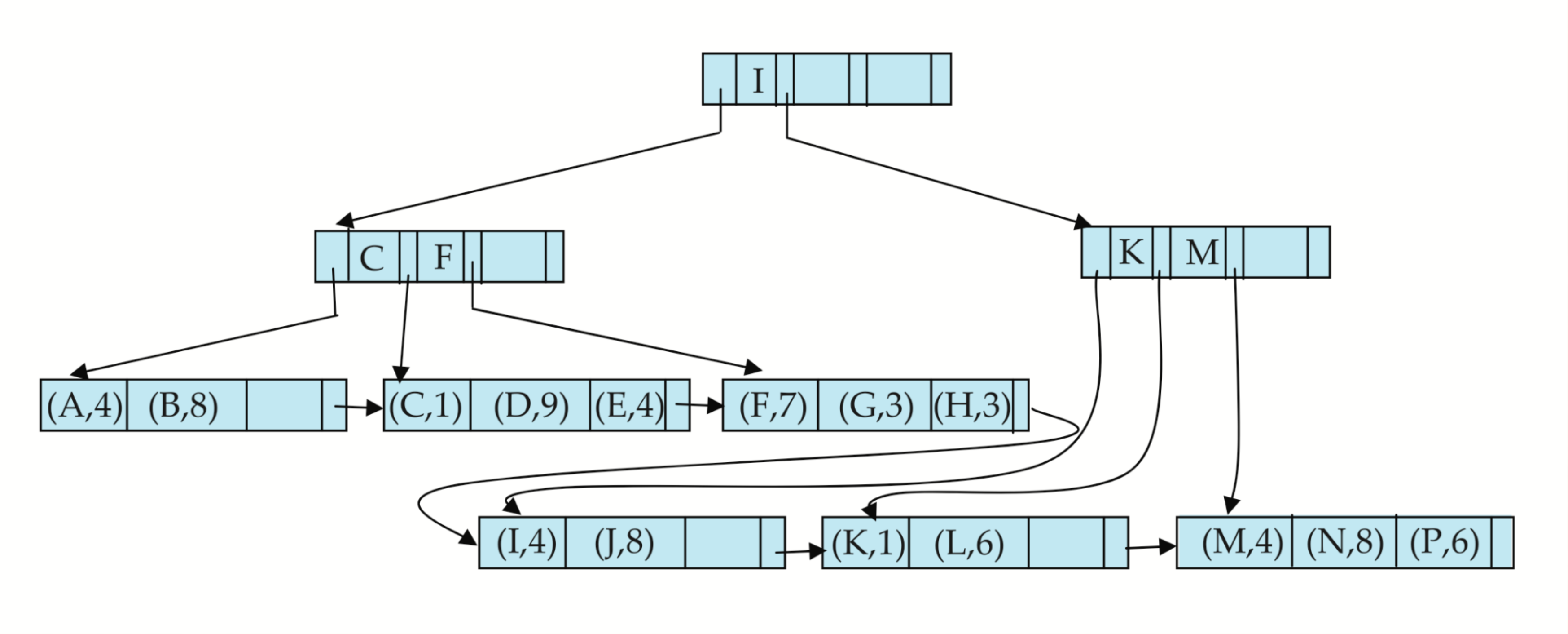
Deletion on B$^+$-Trees

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, 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.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

B^+-Tree File Organization

- Index file degradation problem is solved by using B^+-Tree indices.
- Data file degradation problem is solved by using B^+-Tree File Organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.



Example of B$^+$-tree File Organization

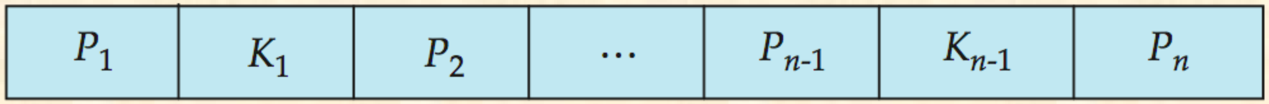
- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

B-Tree Index Files

B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.

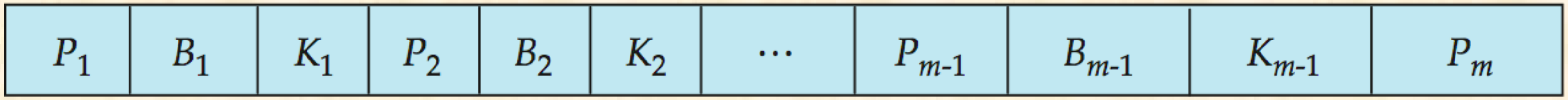
Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.

Generalized B-tree leaf node



P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

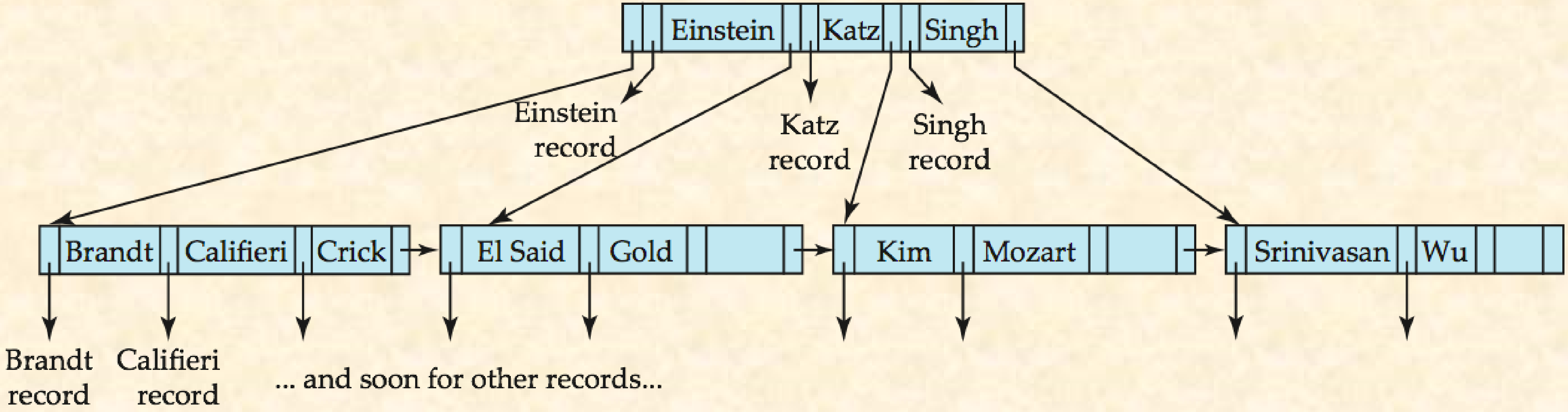
(a)



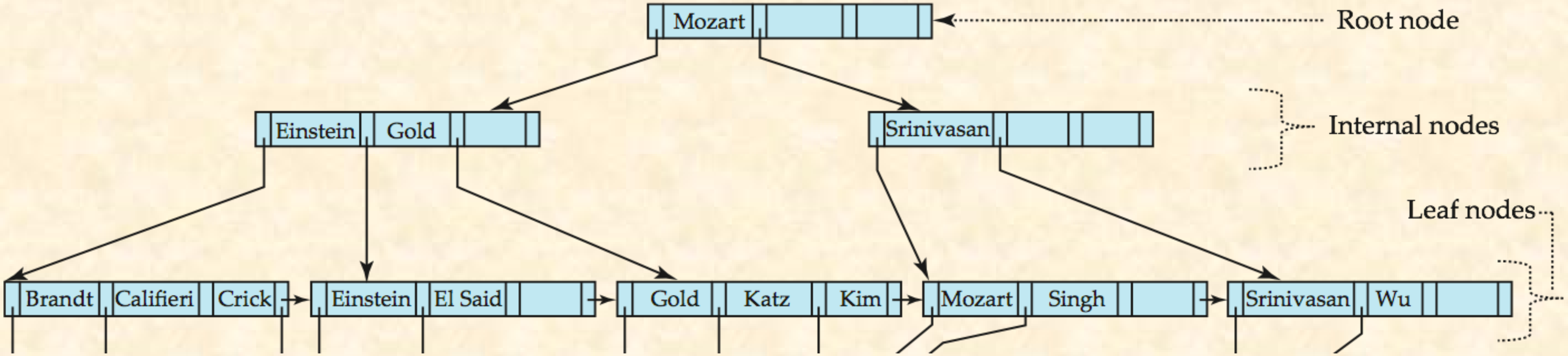
P_1	B_1	K_1	P_2	B_2	K_2	…	P_{m-1}	B_{m-1}	K_{m-1}	P_m

(b)

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



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



B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible 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.

§ 14.5/6 Hashing Index Files

- The file records are stored in a set of *buckets*
    - a **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- **Hash function** *h* ▶
    - a function from the set of all search-key values *K* in a file to the set of all bucket addresses, i.e. the set of the addresses of file records
    - typical hash functions perform computation on the internal binary representation of the search-key.
- **Hash file organization**
    - obtaining the bucket of a record directly from its search-key value using a **hash function**

Sparse Use in Practice！

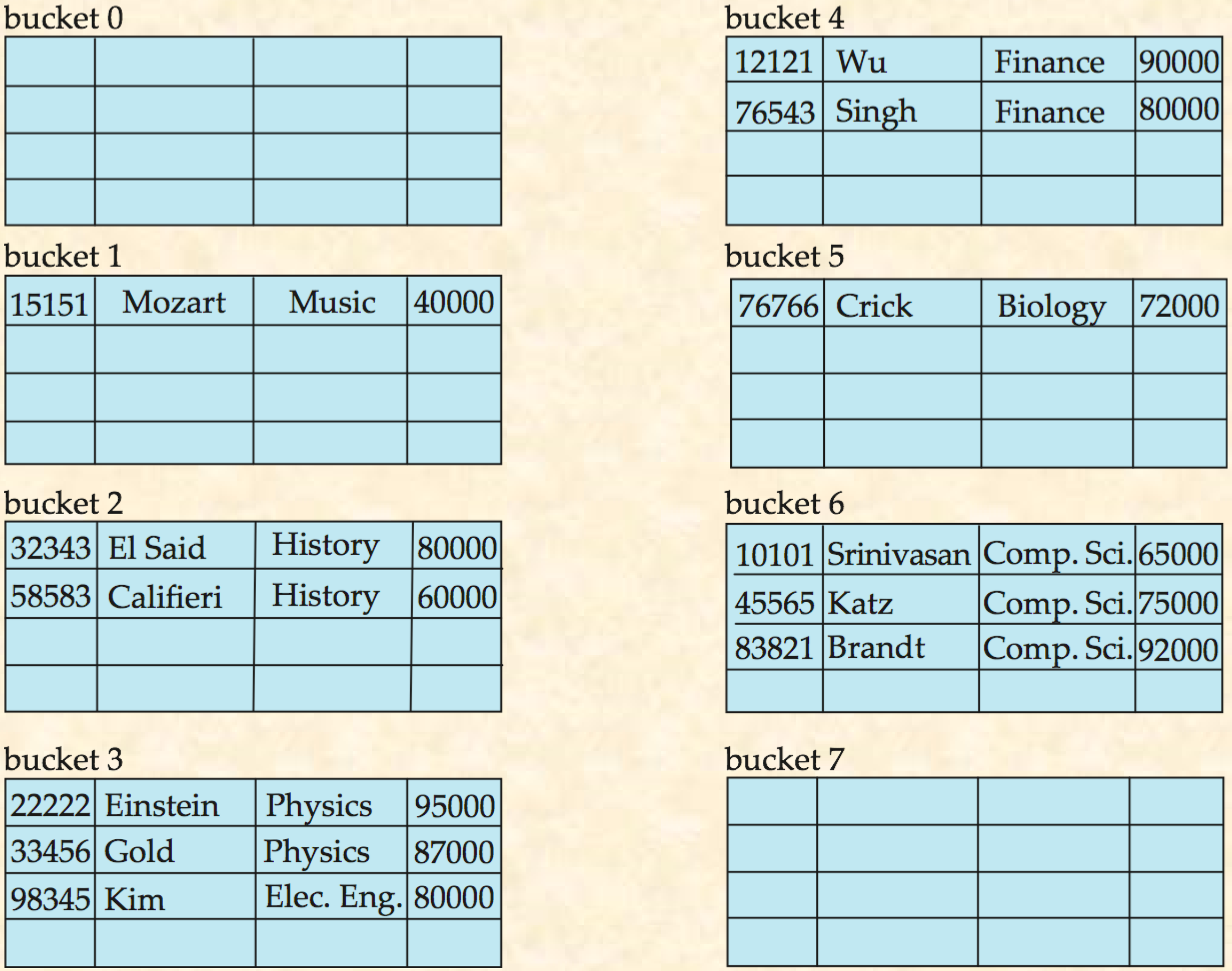
- PostgreSQL supports hash indices, but discourages use due to poor performance
- Oracle supports static hash organization, but not hash indices
- SQLServer supports only B$^+$-trees

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key (See figure in next slide.)

- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3

Example of Hash File Organization



bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

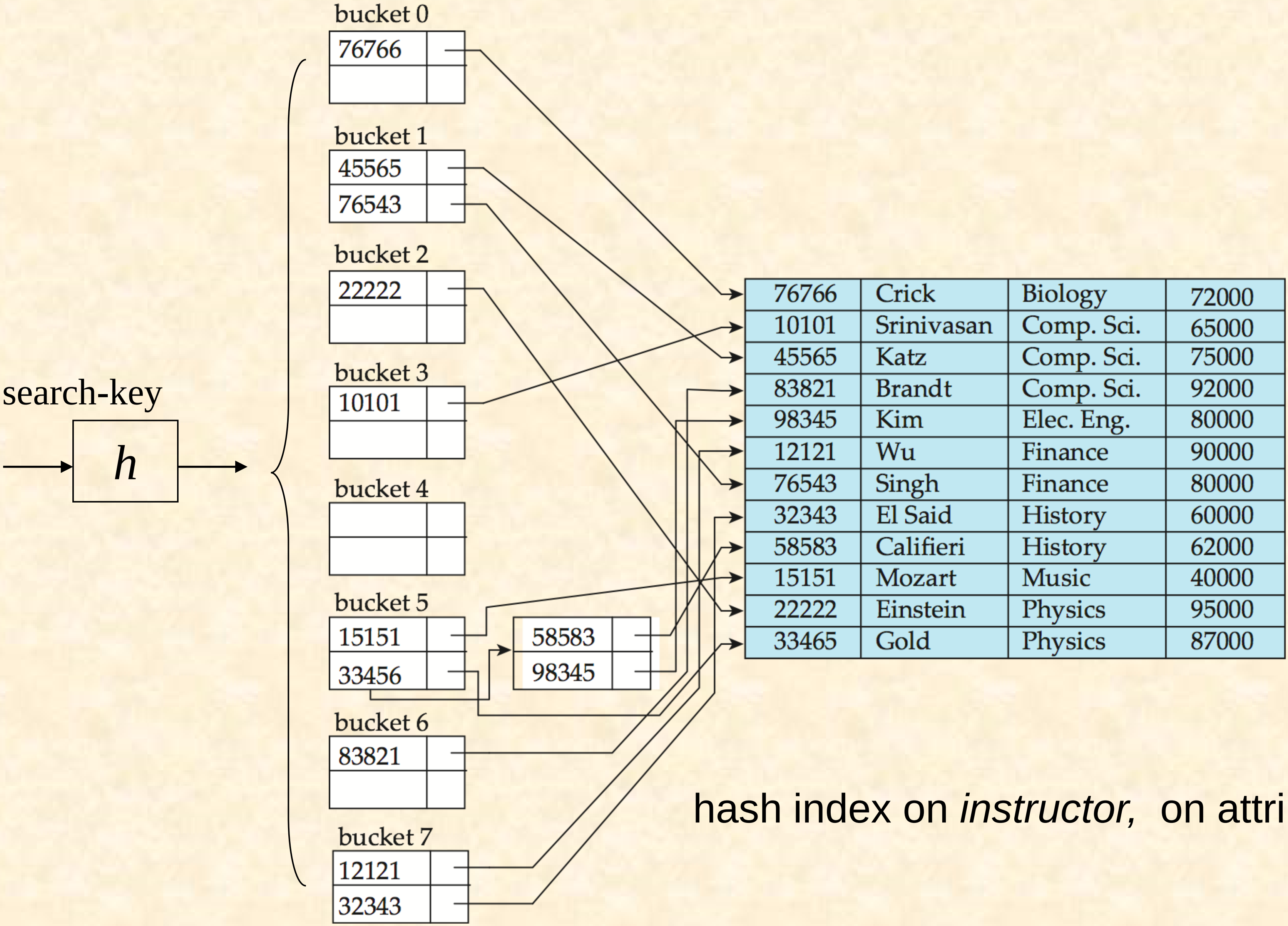
bucket 7

Hash file organization of *instructor* file, using *dept_name* as key (see previous slide for details).

Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index



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

hash index on *instructor*, on attribute *ID*

Static Hashing vs Dynamic Hashing

- Static hashing
  - hash function *h* cannot be modified, while being used

- Dynamic hashing
  - hash function *h* to be modified dynamically

14.8 Comparison of Ordered Indexing and Hashing

- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key, e.g. ***ID=10121***
  - If **range queries** are common, e.g. ***age between 17 and 25***, ordered indices are to be preferred

- 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

§ 14.10 Index Definition and Usage in SQL

- Create an index

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

  E.g.: **create index** *b-index* **on** *branch(branch-name)*

- create cluster index

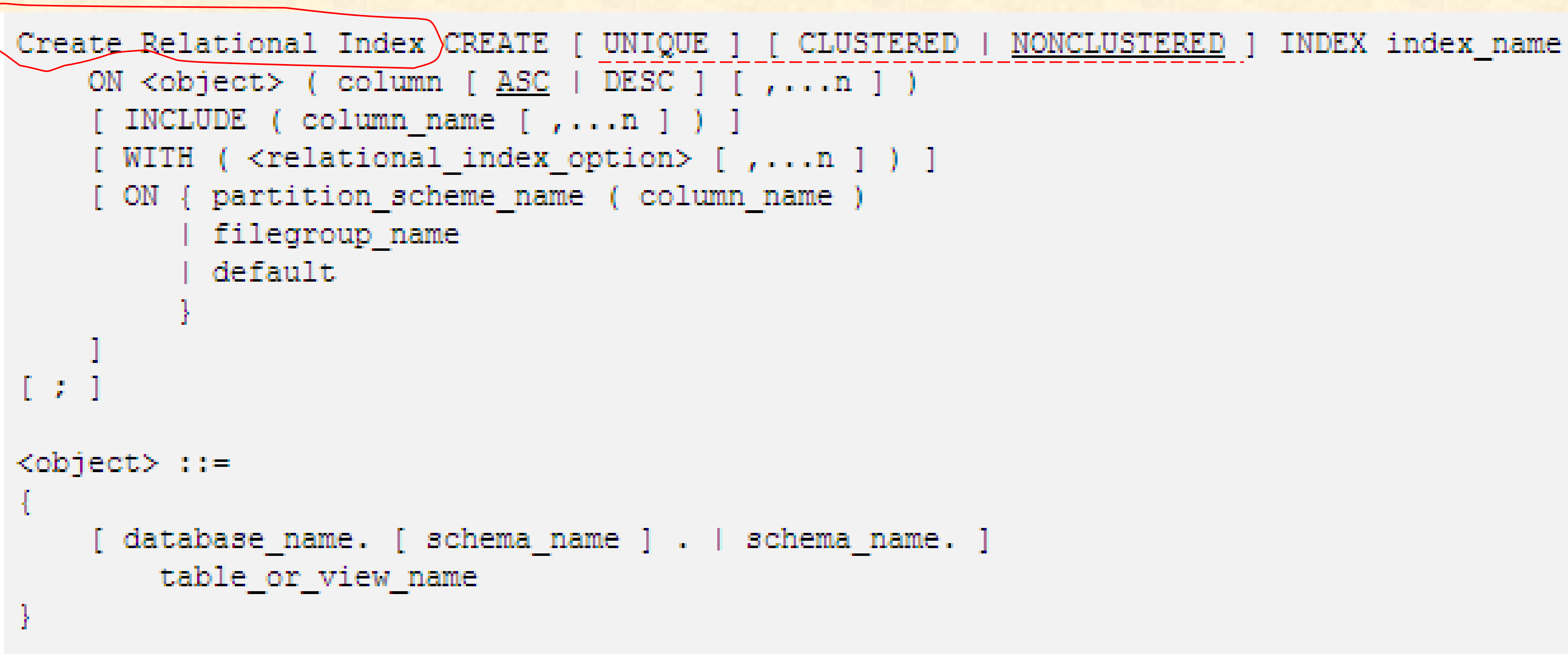
  create noncluster index

  create unique index

- To drop an index

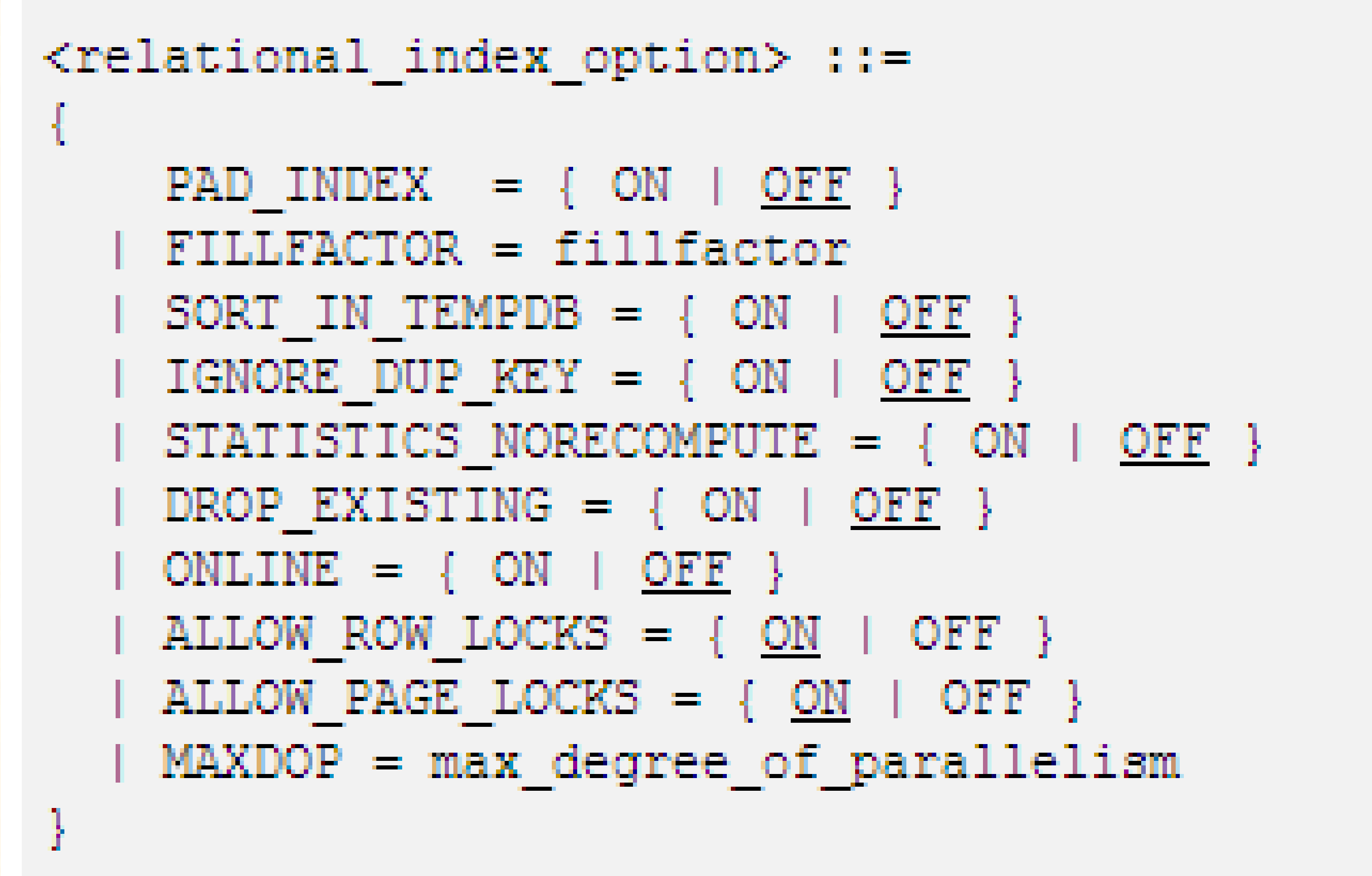
  drop index <index-name>

Create relational index in SQL Server2008

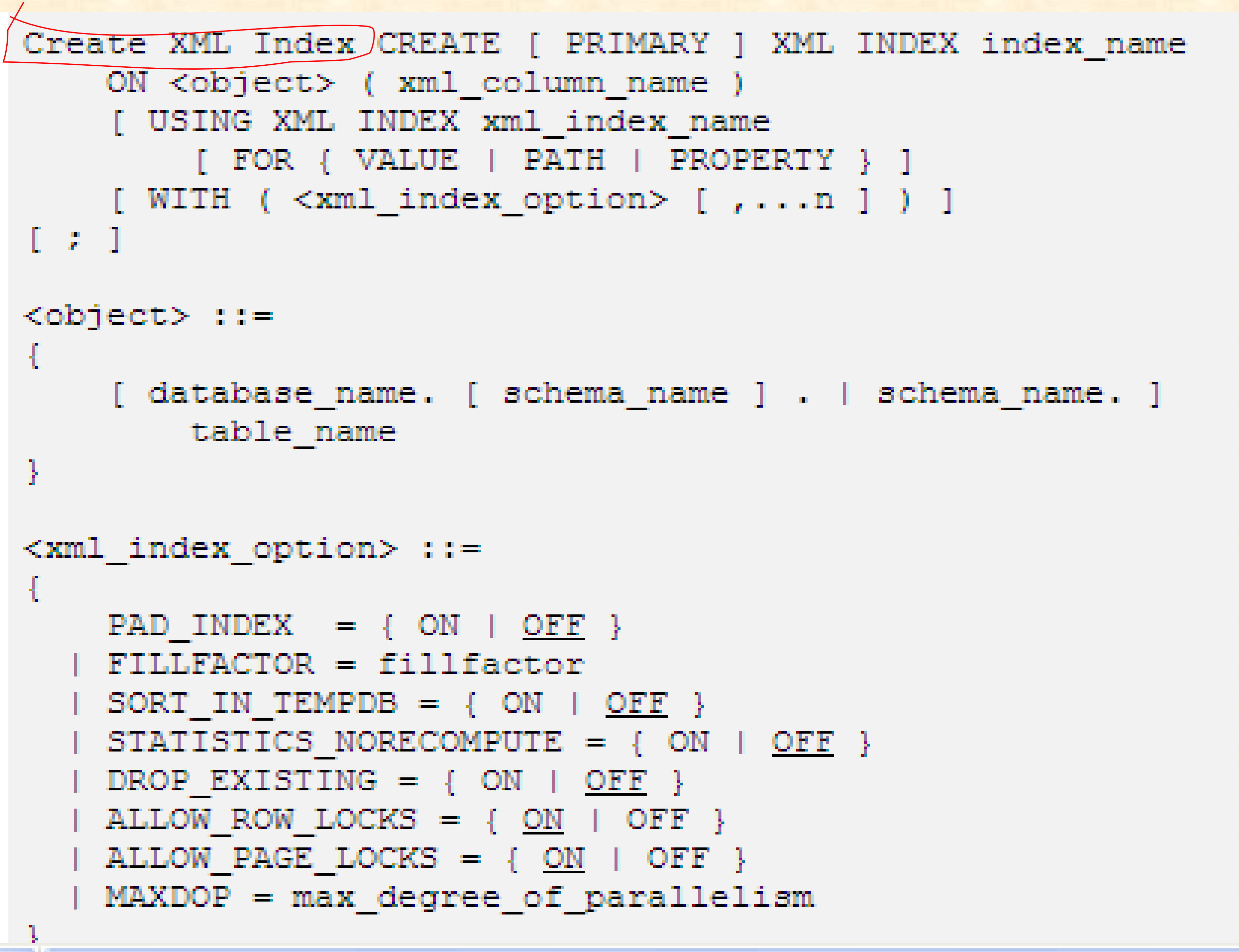


```
Create Relational Index CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    ]
[ ; ]

<object> ::=
{
    [ database_name. [ schema_name ] . | schema_name. ]
        table_or_view_name
}
```



```
<relational_index_option> ::=
{
    PAD_INDEX  = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ONLINE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
}
```



```
Create XML Index CREATE [ PRIMARY ] XML INDEX index_name
    ON <object> ( xml_column_name )
    [ USING XML INDEX xml_index_name
        [ FOR { VALUE | PATH | PROPERTY } ] ]
    [ WITH ( <xml_index_option> [ ,...n ] ) ]
[ ; ]

<object> ::=
{
    [ database_name. [ schema_name ] . | schema_name. ]
        table_name
}

<xml_index_option> ::=
{
    PAD_INDEX  = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
}
```

- **聚集索引**

  对表（堆）创建聚集索引或删除和重新创建现有聚集索引时，要求数据库具有额外的可用工作区来容纳数据排序结果和原始表或现有聚集索引数据的临时副本。有关详细信息，请参阅确定索引的磁盘空间要求。有关聚集索引的详细信息，请参阅创建聚集索引

-

- **唯一索引**

  如果存在唯一索引，数据库引擎 会在每次插入操作添加数据时检查重复值。可生成重复键值的插入操作将被回滚，同时数据库引擎 显示错误消息。即使插入操作更改多行但只导致出现一个重复值时，也是如此。

  如果在存在唯一索引并且 IGNORE_DUP_KEY 子句设置为 ON 的情况下输入数据，则只有违反 UNIQUE 索引的行才会失败。有关唯一索引的详细信息，请参阅创建唯一索引。