# Topic 4: File Organization, Indexing and Hashing (Chapters 13, 14, 24)

**Database System Concepts**


**(Modified for CS 4513)**

# Topic 4: File Organization, Indexing and Hashing

- How to represent records in a file structure
- How to organize records in a file: different file organizations
  - Heap Files
  - Sequential Files
  - Indexed Sequential Files
  - B-Tree Index Files
  - B+-Tree Index Files
  - Multiple-Key Access Using Indices
  - Static Hashing
  - Dynamic Hashing
  - Comparison of Ordered Indexing and Hashing
  - Multi-table Clustering File Organization
- Index Definition in SQL
- Factors Considered in Deciding a File Organization for a Table
- Data Dictionary Storage

# Query Processing Flow Diagram

# How Records Are Represented in a File

- The database is stored as a collection of *files*. Each file is a sequence of *records.* A record is a sequence of fields.
- One approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

  This case is easiest to implement; will consider variable length records later.

# Fixed-Length Records

- Simple approach:
  - Store record *i* starting from byte $n * (i - 1)$, where *n* is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries
- Deletion of record *i:* alternatives*:*
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record *n* to *i*
  - do not move records, but link all free records on a *free list*

| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

# Deleting record 3 and compacting

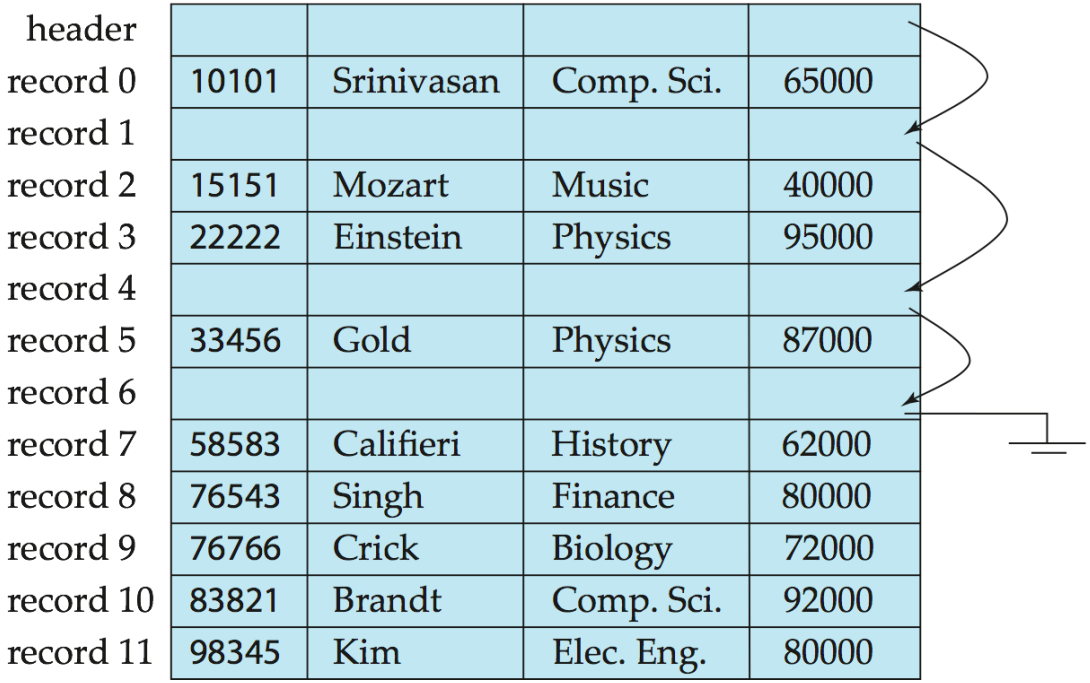
| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

# Deleting record 3 and moving last record

| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |

# Free Lists

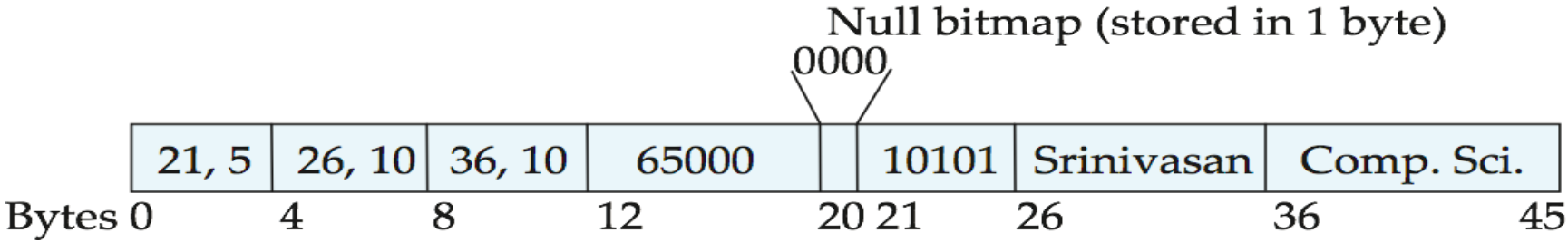
- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)



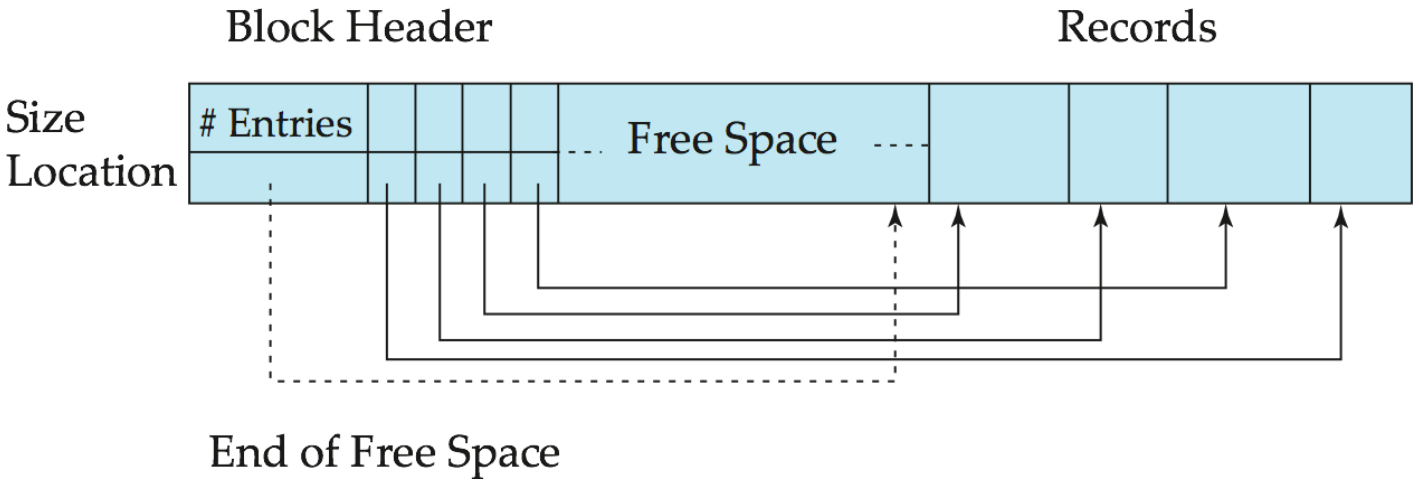
| | | | | |
|---|---|---|---|---|
| header | | | | |
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | | | | |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | | | | |
| record 5 | 33456 | Gold | Physics | 87000 |
| record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

# Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (e.g. arrays)
- Issues:
  - How to represent a single record in such a way that individual attributes can be extracted easily
  - How to store variable-length records within a block, such that records in a block can be extracted easily
- Variable length attributes are represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values are represented by null-value bitmap



# Variable-Length Records: Slotted Page Structure




- **Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.

# How to Organize Records in A File: File Organization

- Goal: design/implement an efficient file organization to improve query processing (to improve query response time and system throughput)

# Heap File (Pile File)

- Records are placed in a file in the order in which they are inserted
- New records are inserted at the end of the file
- Efficient in insertion (no sorting is involved)
- Slow search due to linear search
- Use it when we want to populate the table for future use, but we do not know how we will use this table (should be tuned later after monitoring query response time)

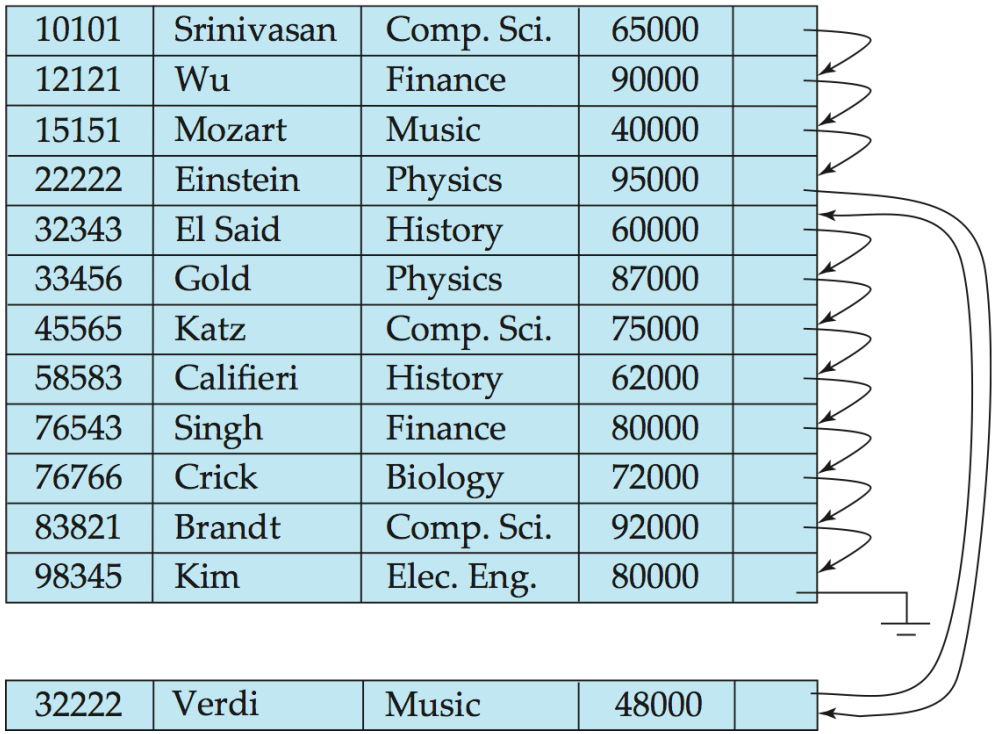
# Sequential File (Ordered File) Organization

- Records are placed in the file in a search-key order
- Suitable for applications that require sequential processing of records in sorted order based on some search key
- Slow insertion, why?

| | | | | |
|---|---|---|---|---|
| 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 | |

# Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



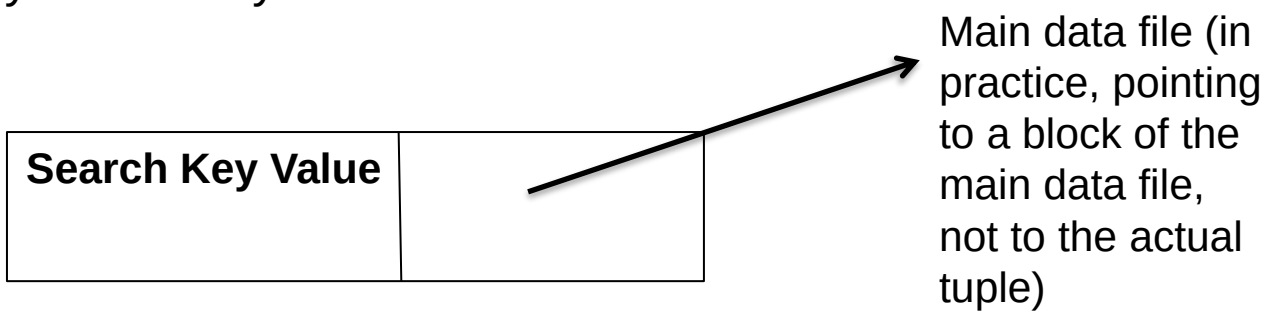
| | | | | |
|---|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 | |
| 12121 | Wu | Finance | 90000 | |
| 15151 | Mozart | Music | 40000 | |
| 22222 | Einstein | Physics | 95000 | |
| 32343 | El Said | History | 60000 | |
| 33456 | Gold | Physics | 87000 | |
| 45565 | Katz | Comp. Sci. | 75000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

| | | | | |
|---|---|---|---|---|
| 32222 | Verdi | Music | 48000 | |

# Indexed-Sequential File

- Main data file: a sequential file storing complete tuples sorted on a search key.
- Index: Primary index and Secondary index; Index files are typically much smaller than the original file
  - Primary index: index built on the search key of the main data file (the search key of a primary index is usually but not necessarily the primary key) (also called clustering index)
  - Secondary index: index built on the other attributes in the main data file (also called non-clustering index)
  - Structure of index:

    Primary index entry:



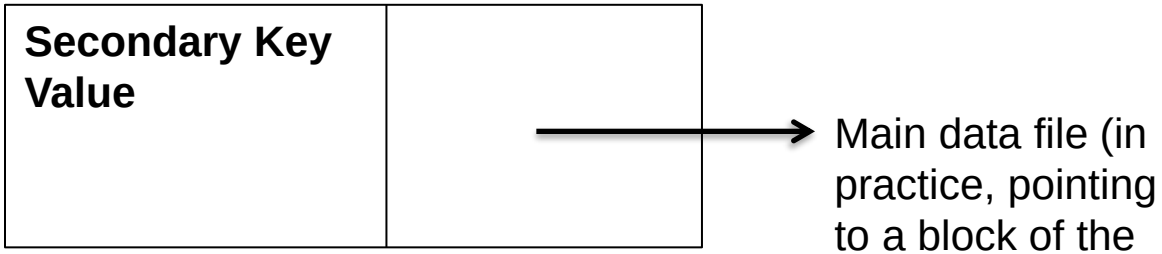

 

# Indexing-Sequential File (cont.)

- Example of Primary Index

# Indexed-Sequential File (cont.)

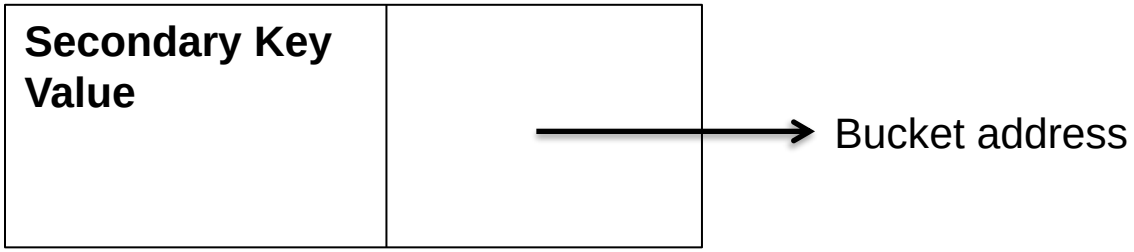
- Indexed-Sequential file
  - Secondary index: based on secondary key value (non-ordering field)
  - Secondary key is a candidate key (e.g. licence_no)




Example:

# Indexed-Sequential File (cont.)

- Indexed-Sequential file
  - Secondary key is a non-candidate key (not unique, e.g. car_color)




Example:

# Indexed-Sequential File (cont.)

- Example: Using the file organizations created in the previous example, show steps needed to answer the following query: “Find all drivers whose license number is between 20 and 50.”

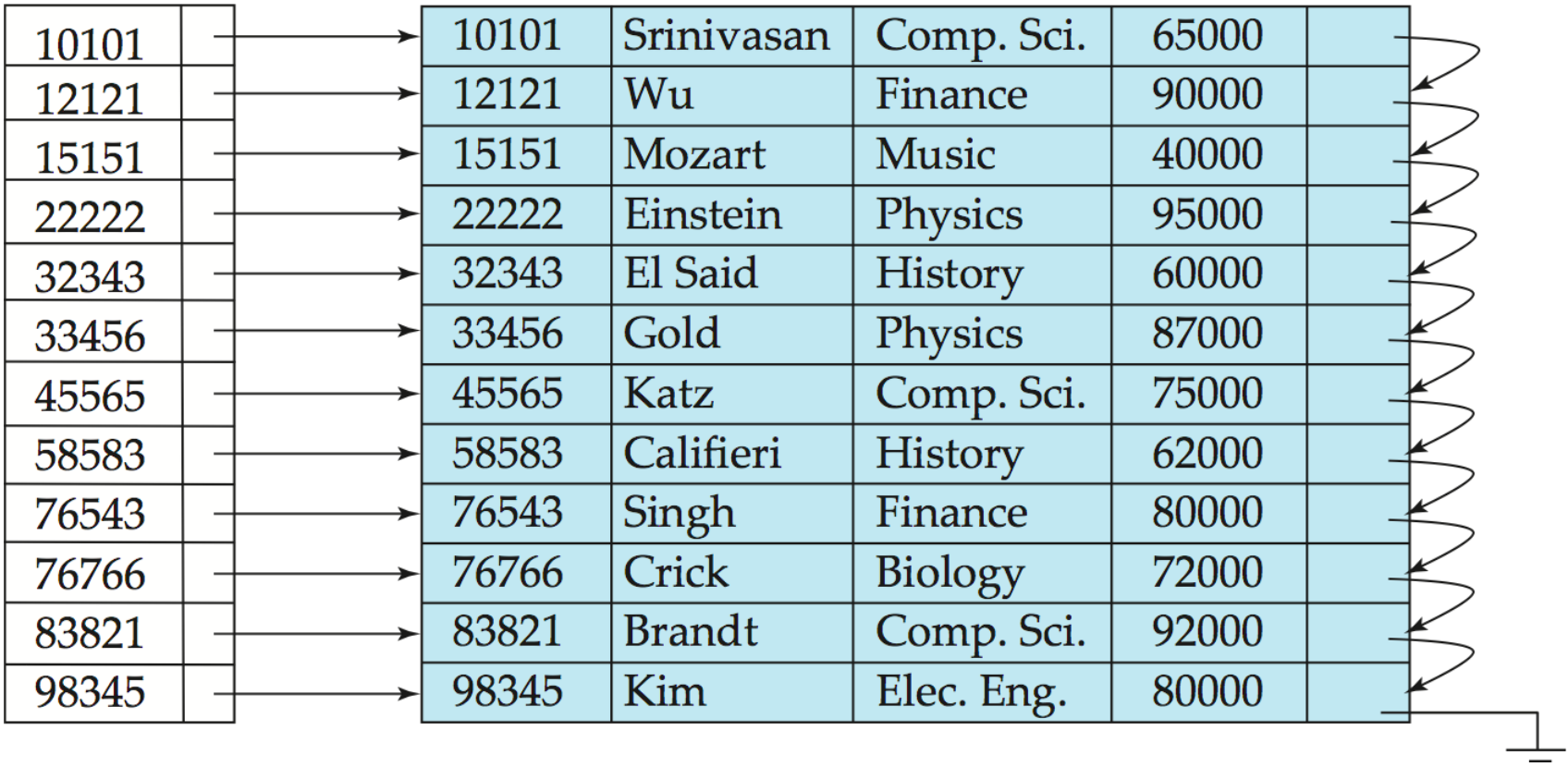
- Answer:

# Indexed-Sequential File (Cont.)

- Dense Index
- Sparse Index

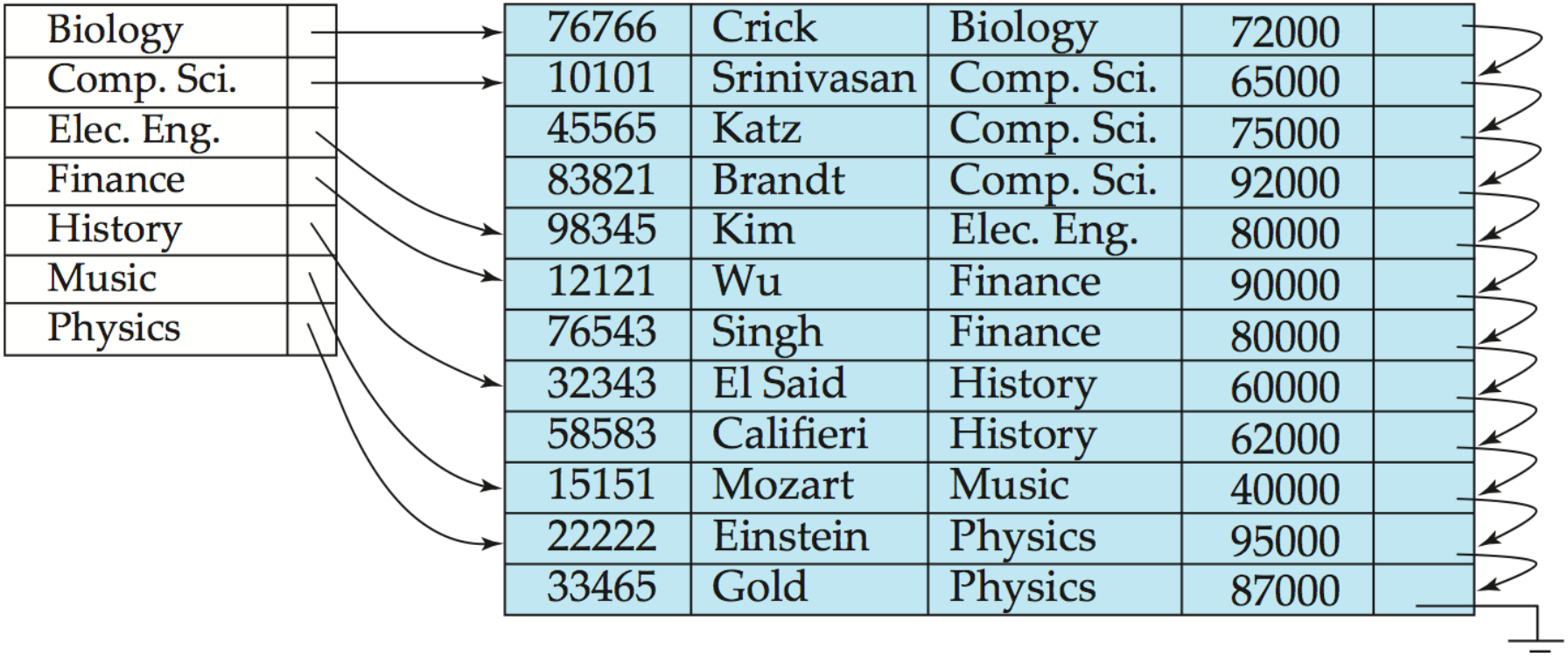
# Dense Index Files

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



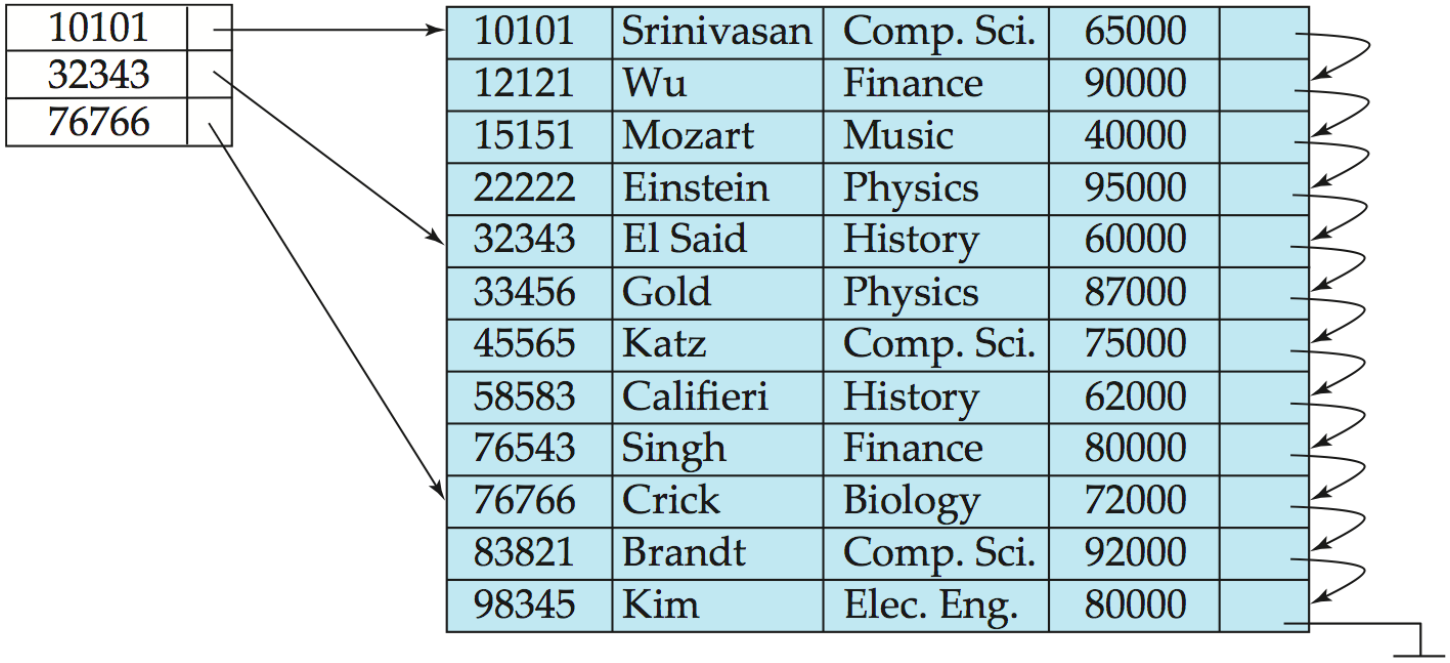
# Dense Index Files (Cont.)

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



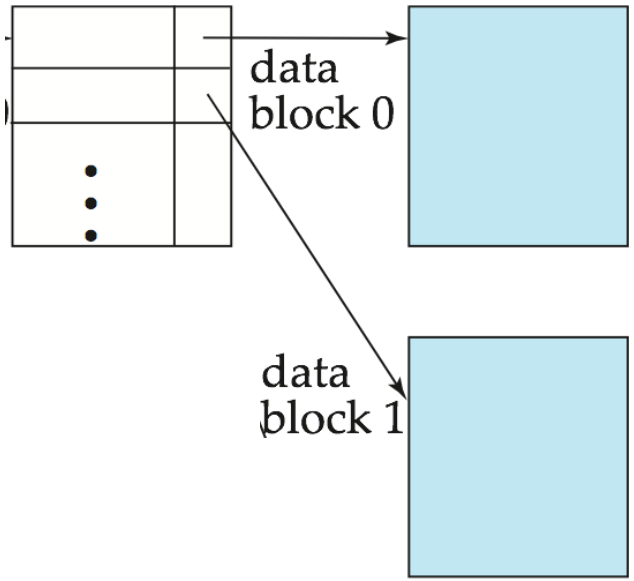
# Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - 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

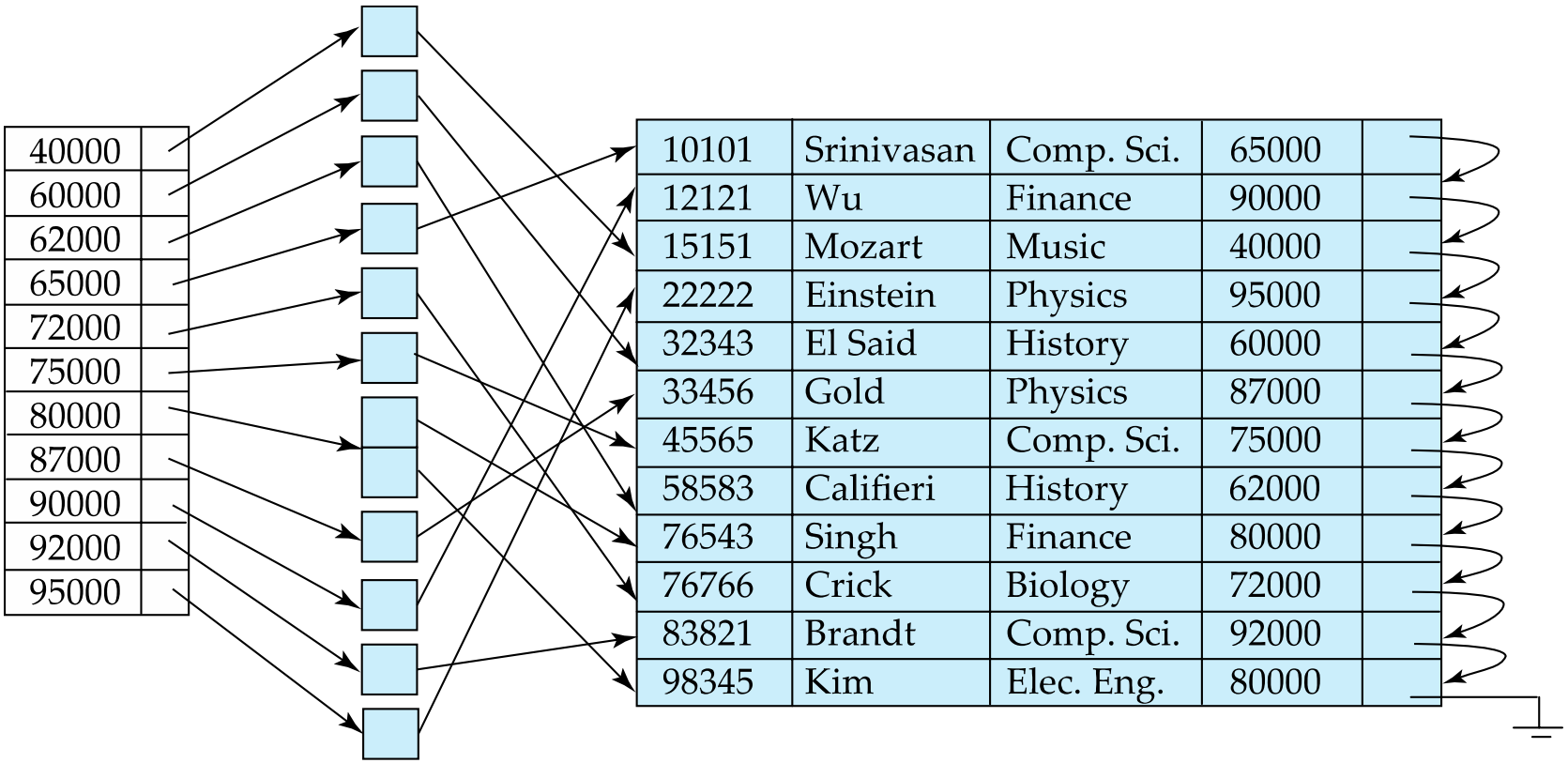


# Sparse Index Files (Cont.)

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



# 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, why?
- How about primary indices?

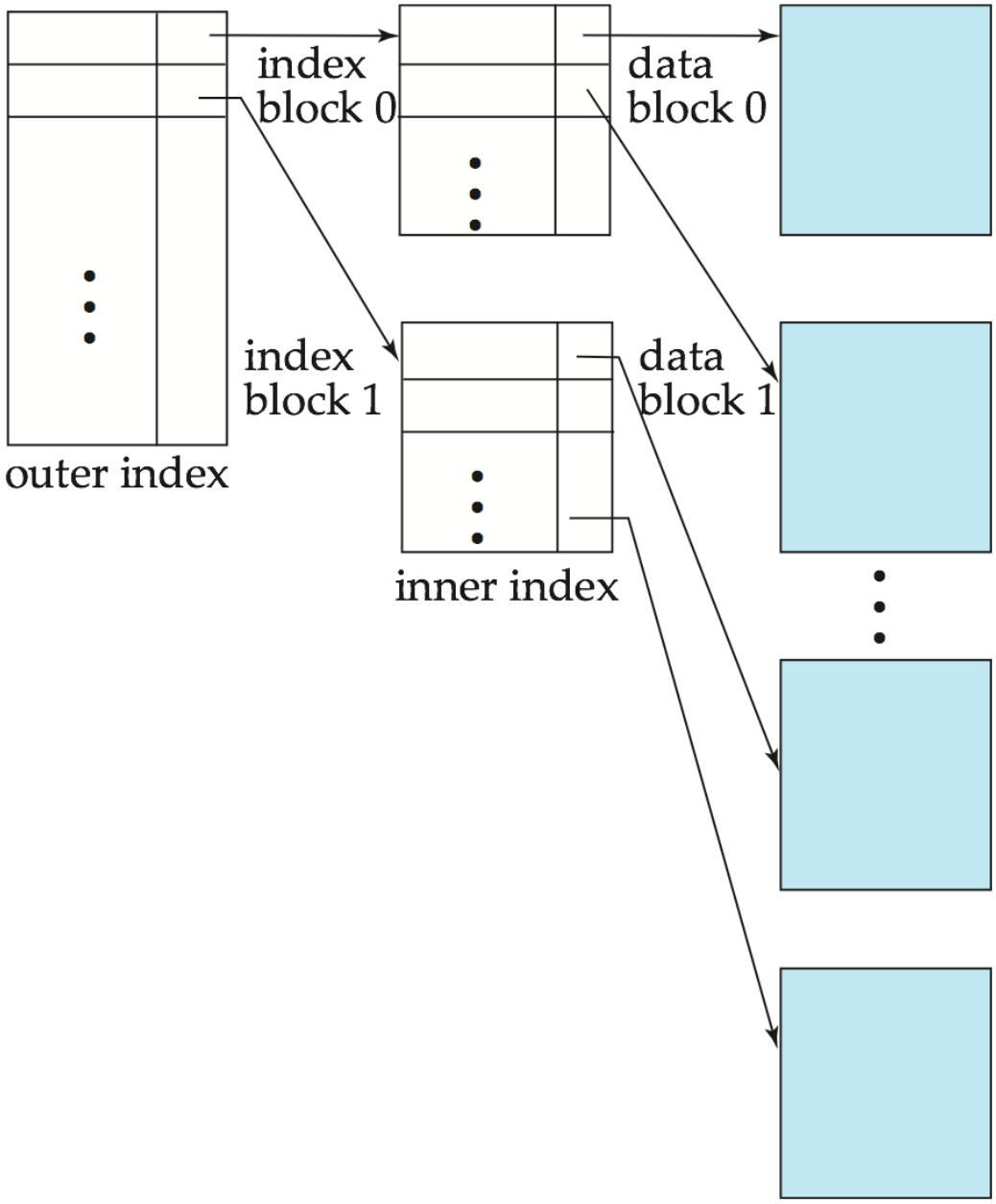
# Primary and Secondary Indices

- 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, versus about 100 nanoseconds for memory access

# Multilevel Index

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

# Multilevel Index (Cont.)



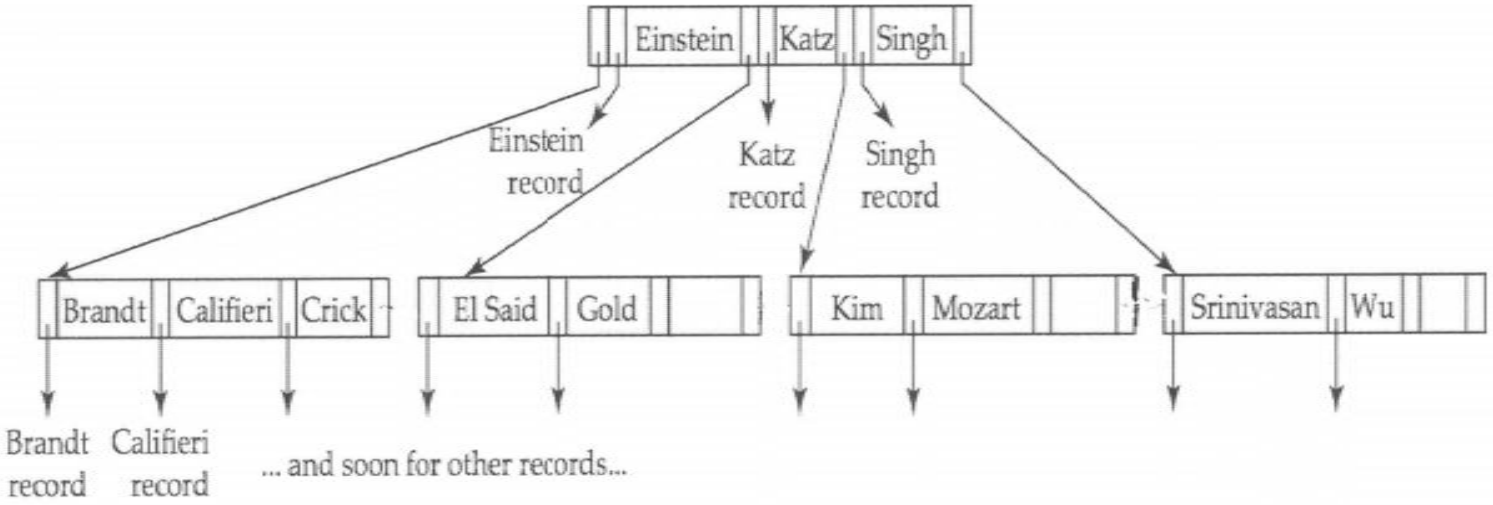
# Indexed Sequential Files (Cont.)

- Facilitates sequential search on search keys
- Performance degrades as the file grows, both for index lookups and for sequential scans through the data

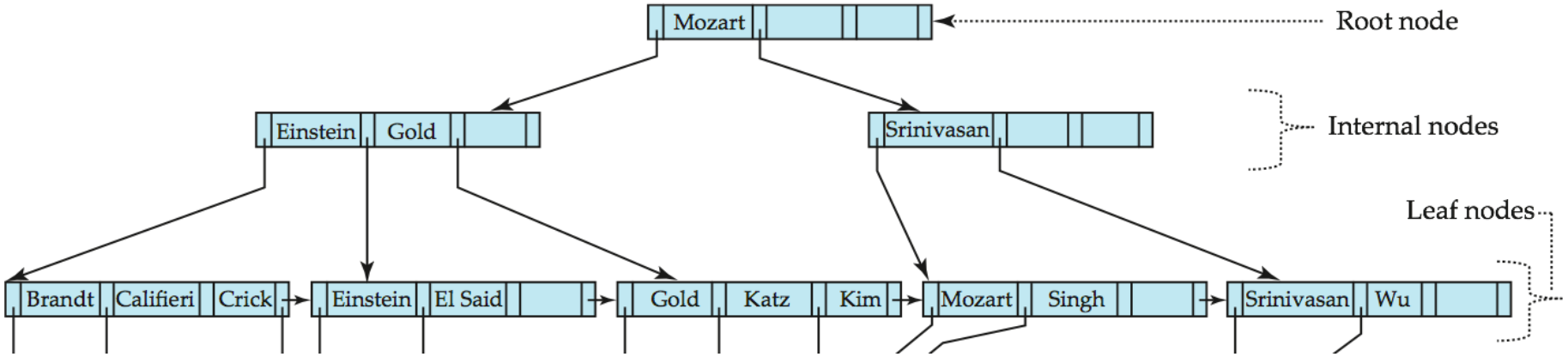
# B-Tree and $B^+$ -Tree Indices

- Work well despite insertions and deletions

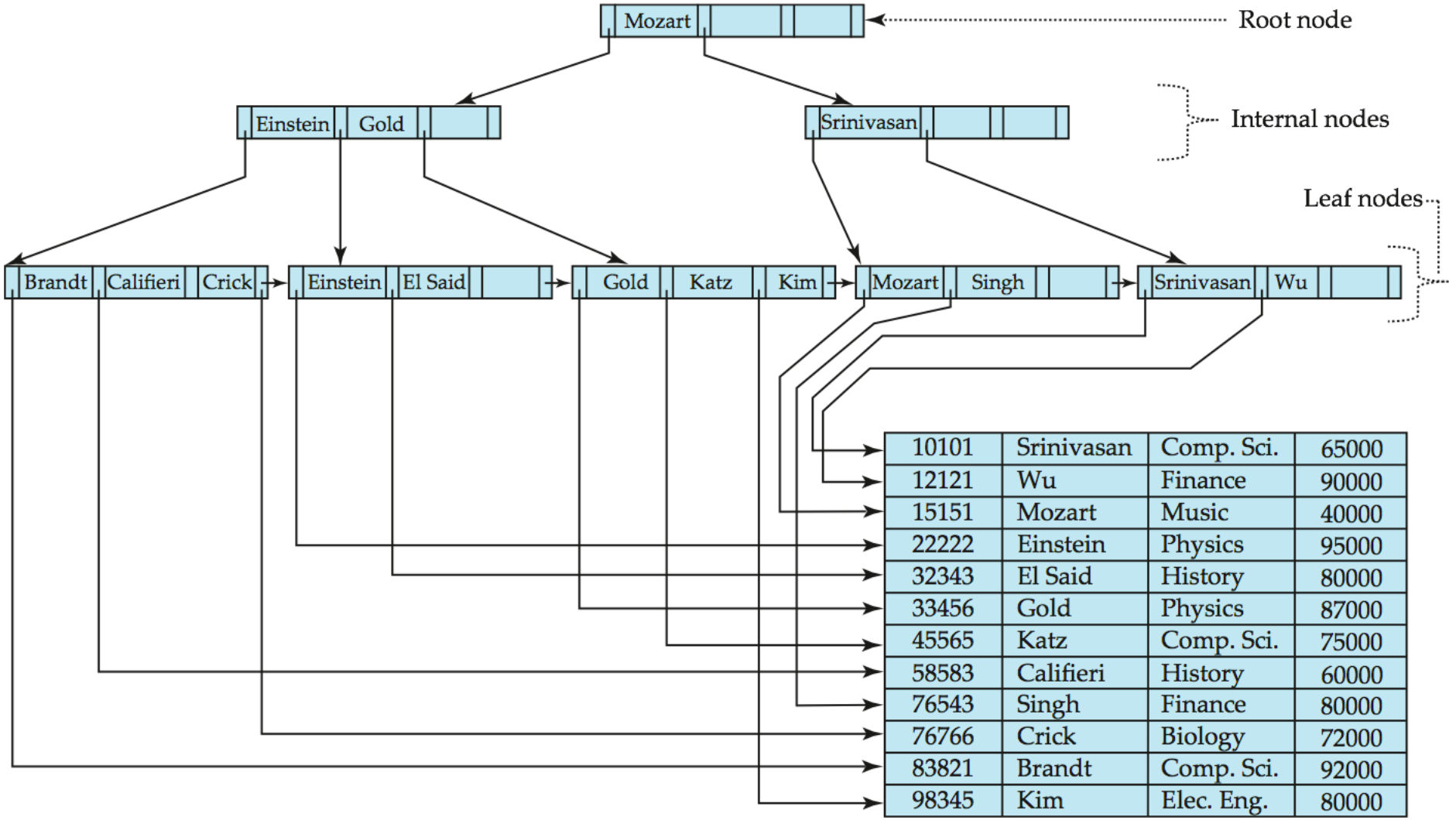
# B-Tree and B+ -Tree Index File Examples




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



# Example of B+ -Tree (Index and Main Data)

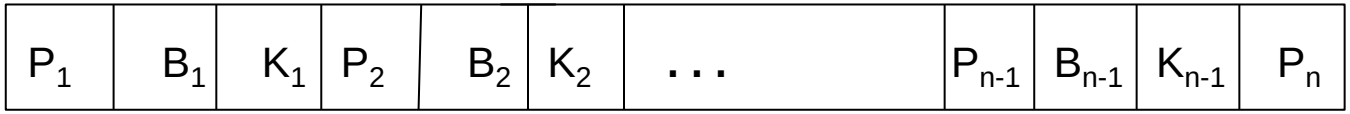



| | | | |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 80000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 58583 | Califieri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

# B-Tree Index Structure

- B-Tree Index of order n
- A balanced tree
- Root has at least 2 children unless the tree has only 1 node
- A leaf or non-leaf node (except root) contains at least ($\lceil n/2 \rceil - 1$) search key values and at most (n-1) search key values
- A non-leaf node with k children contains (k-1) search key values
- Structure of a leaf node:


- Structure of a non-leaf node:



# B-Tree Insertion Procedure

- Fill up root (level 0)
- When root is full, split it into 2 nodes at level 1
  - Keep the middle value at the root (level 0)
  - Split the rest of the values as evenly as possible and store them in the two nodes at level 1
- If a non-root node is full, split the node into 2 nodes at the same level
  - Move the middle value to the parent node
  - If the parent node is full, recursively apply the procedure

# B-Tree Insertion Example

# B-Tree Deletion Procedure

- If deletion of value $K_i$ in a non-leaf node causes a node to be too small
  - Move the smallest key of the sub-tree pointed to by pointer $P_{i+1}$ to the place of $K_i$
- If $K_i$ is a leaf node, then redistribute the search key values over sibling leaf nodes and adjust the pointers along the path from leaf to root. This in turn may cause redistribution of search keys in non-leaf nodes

# B-Tree Deletion Example

# $B^+$-Tree Index Structure

- A search key value may appear more than once
- Data pointers are sored at leaf nodes only
- Every search key value appears once at the leaf level
- Leaf nodes are linked together to facilitate ordered access on search key values
- Efficient for random search and range search
- Structure of a non-leaf node:

- Structure of a leaf node:

# $B^+$ -Tree Index Insertion Procedure

- Want to insert key $K_i$ into a B+ -tree
- Search for a leaf node $L_i$ that is supposed to store Ki
- If $L_i$ has (n-1) values, then it is FULL
  - Split $L_i$ into 2 nodes
  - Put ($\lceil n/2 \rceil$-1) values into existing node
  - Put the rest of the values into the new nodes
  - Put the smallest value in the new node into the parent
  - If parent is full, recursively split it and adjust the pointers

# $B^+$ -Tree Index Insertion Example

# $B^+$ -Tree Index Deletion Example

# B-Tree Index vs. $B^+$-Tree Index

- Advantages of B-Tree indices:
  - May use fewer tree nodes than a corresponding $B^+$-Tree.
  - Sometimes possible to find the search-key value before reaching leaf nodes.
- Disadvantages of B-Tree indices:
  - Only a 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 are more complicated than in $B^+$-Trees
  - Implementation is harder than $B^+$-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

# Multiple-Key Access Using Indices on Single Attributes

- 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.

# Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B*
  - Hash function h on search key k: h(k): bucket address
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

# Example of Hash File Organization

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

- There are 8 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 8
  - E.g. h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3
  - Music = 13+21+19+9+3 = 65
  - h(Music)=65 modulo 8 = 1

# 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

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

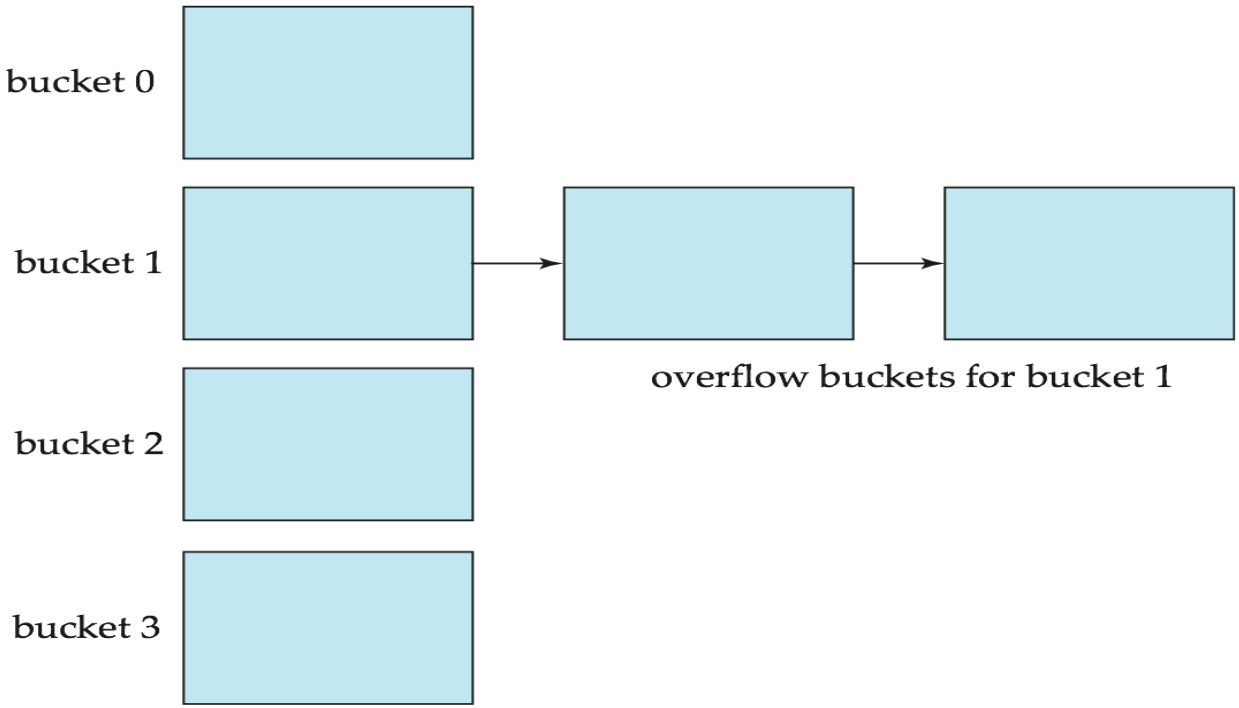
Hash file organization of *instructor* file, using *dept_name* as key (see previous slide for details).

# Static Hashing

- Is it good for range search or random search?
- What is the worst hash function?
- What is an ideal hash function?
- When do bucket overflows happen?

# Handling of Bucket Overflows

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.



# 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 the initial number of buckets is too small, and file grows, performance will degrade due to too many overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underutilized).
  - 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.

# Extendable Hashing

- Dynamically adjust the hash function as the database size grows/shrinks
- The same hash function can be used
- The number of buckets will grow/shrink as needed
- Example:

# Extendable Hashing Example (cont.)

# Extendable Hashing (Cont.)

- What are its advantages?

- What are its disadvantages?

# Comparison 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.
  - If range queries are common, ordered indices are to be preferred

# Multitable Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization

*department*

| *dept_name* | *building* | *budget* |
|---|---|---|
| Comp. Sci. | Taylor | 100000 |
| Physics | Watson | 70000 |

*instructor*

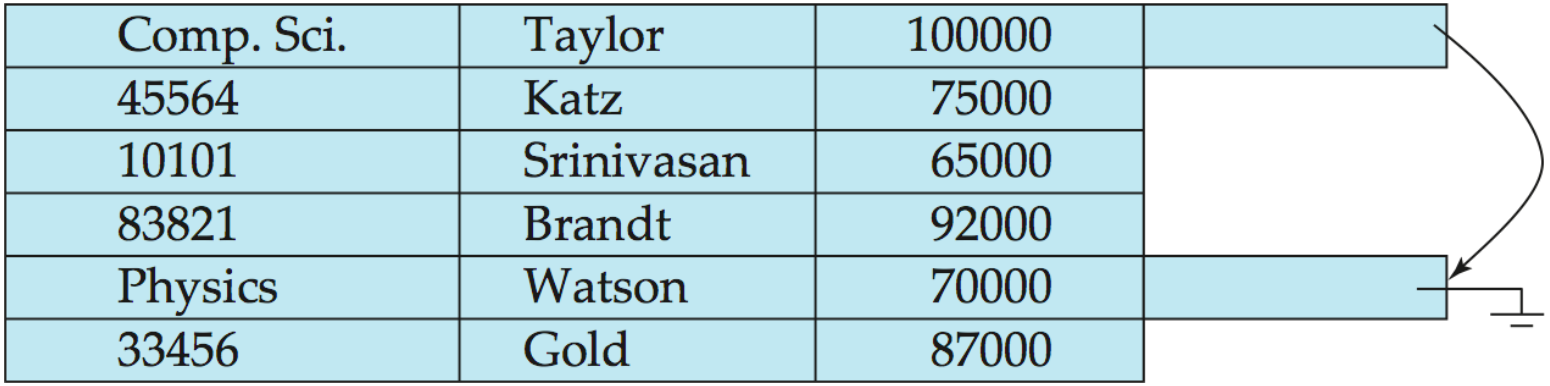
| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |

multitable clustering of *department* and *instructor*

| | | |
|---|---|---|
| Comp. Sci. | Taylor | 100000 |
| 45564 | Katz | 75000 |
| 10101 | Srinivasan | 65000 |
| 83821 | Brandt | 92000 |
| Physics | Watson | 70000 |
| 33456 | Gold | 87000 |

# Multitable Clustering File Organization (cont.)

- good for queries involving *department* ⋈ *instructor*, and for queries involving one single department and its instructors
- bad for queries involving only *department*
- results in variable size records
- Can add pointer chains to link records of a particular relation



| | | | |
|---|---|---|---|
| Comp. Sci. | Taylor | 100000 | |
| 45564 | Katz | 75000 | |
| 10101 | Srinivasan | 65000 | |
| 83821 | Brandt | 92000 | |
| Physics | Watson | 70000 | |
| 33456 | Gold | 87000 | |

# 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

  **drop index** <index-name>

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

# Factors Considered When Deciding on a File Organization for a Table

- Types of queries on the table
  - Random search
  - Range search
  - Insertion
  - Deletion
- Frequencies of queries
- Access time
- Insertion time
- Deletion time
- Space overhead

# Factors Considered When Deciding on a File Organization for a Table - Example

- Example: Given the following tables and queries, decide an appropriate file organization for each table.

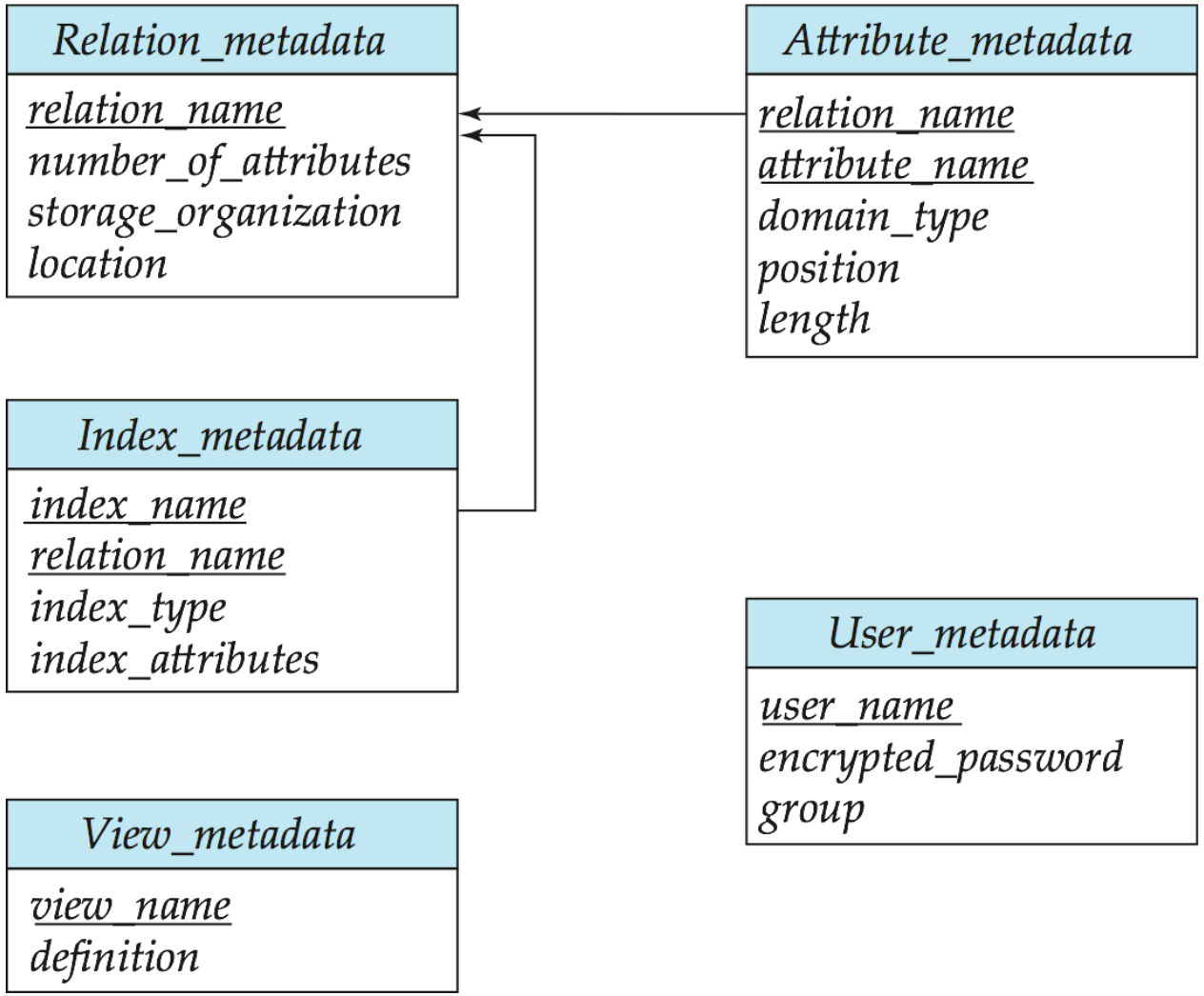
# Data Dictionary Storage

The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- Information about relations
  - names of relations
  - names, types and lengths of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/…)
  - Physical location of relation
- Information about indices

# Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory



# End of Topic 4

**Database System Concepts**