# Chapter 13: Data Storage Structures

Sistemas de Bases de Dados 2019/20

Capítulo refere-se a: Database System Concepts, 7th Ed

# File Organization

- Both magnetic disks and SSD are block structure (with blocks of 4K~8K)
- Databases are structures in tables with records.
- DBMS (as well as OS) consider *files* to abstract the details of stored blocks
  - But it is useful to be aware of the block structure for efficiency

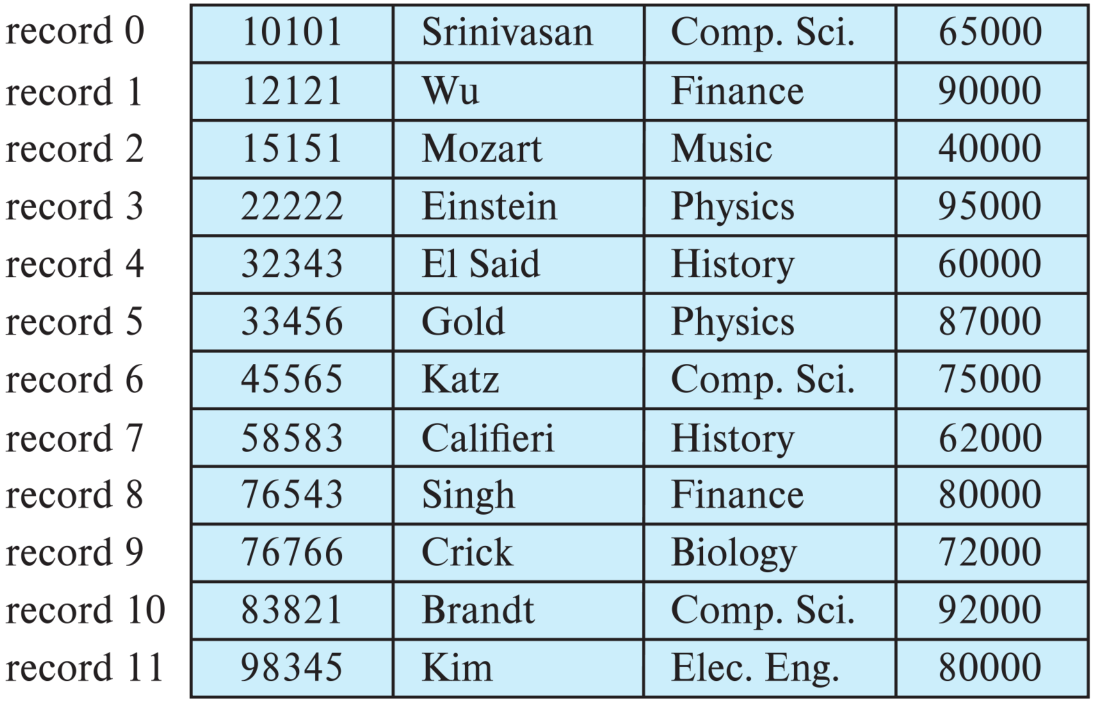
- The database is stored as a collection of *files*. Each file is a sequence of *records.* A record is a sequence of fields.
- First (naïve) approach
  - Assume record size is fixed
  - Each file has records of one particular type only
  - Different files are used for different relations

  This case is the easiest to implement; we consider variable length records later
- We start by assuming that records are smaller than a disk block

.

# 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 easy. But records may cross blocks!
    - Modification: do not allow records to cross block boundaries

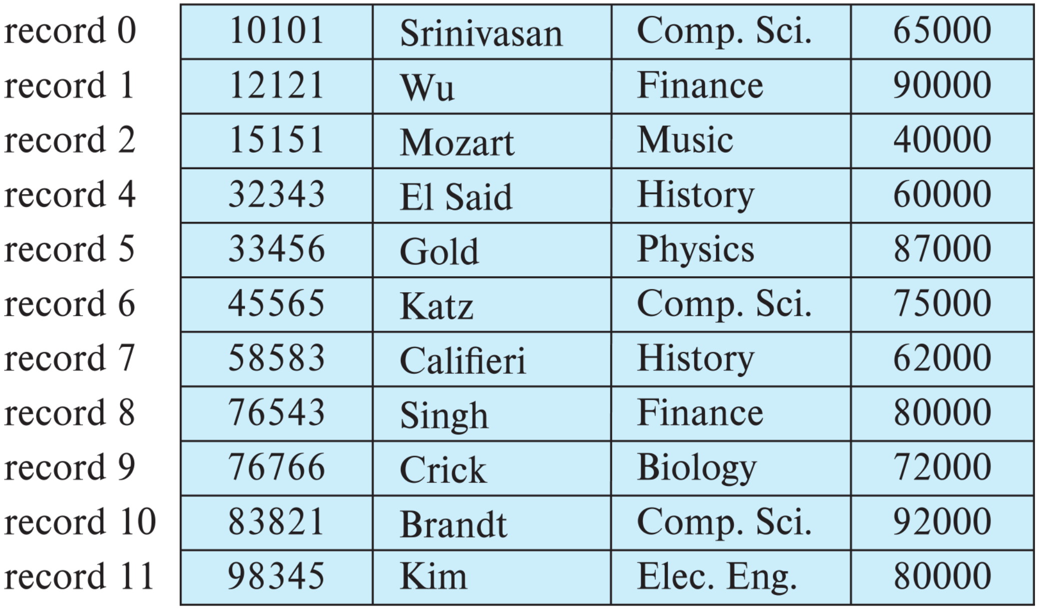


| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | 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 |

# Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - **move records *i* + 1, . . ., *n* to *i, . . . , n* – 1**
  - move record *n* to *i*
  - do not move records, but link all free records on a *free list*

**Record 3 deleted**

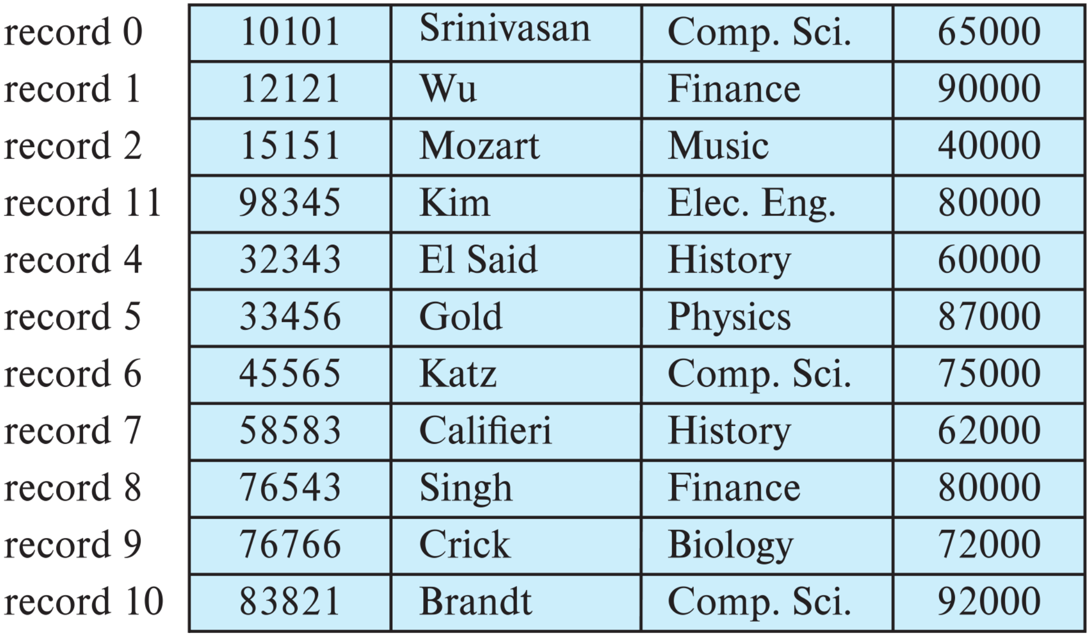


| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | 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 |

# Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - move records *i* + 1, . . ., *n* to *i, . . . , n* – 1
  - **move record *n* to *i***
  - do not move records, but link all free records on a *free list*

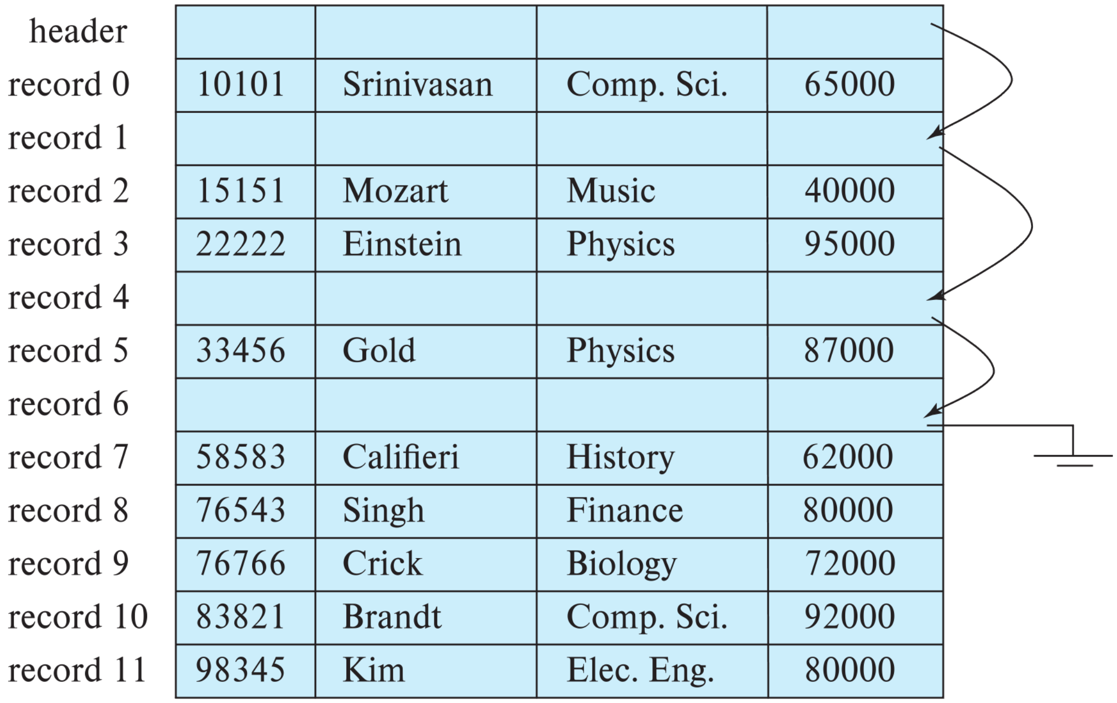
**Record 3 deleted and replaced by record 11**



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

# Fixed-Length Records

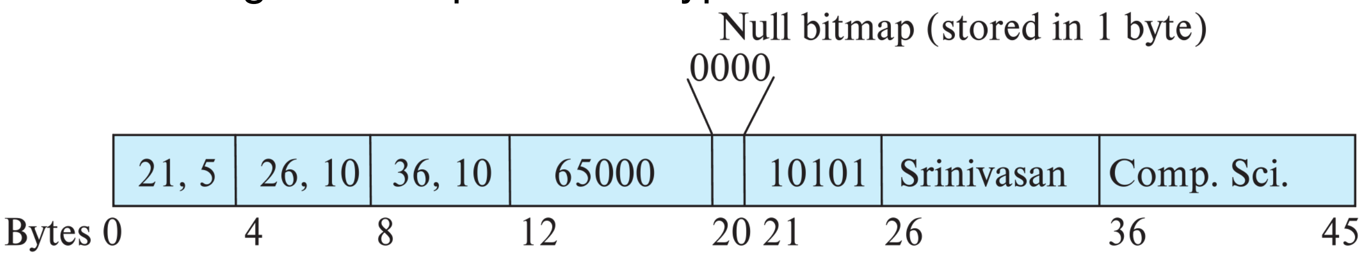
- Deletion of record *i:* alternatives*:*
  - move records *i* + 1, . . ., *n* to *i, . . . , n* – 1
  - move record *n* to *i*
  - **do not move records, but link all free records on a *free list***



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

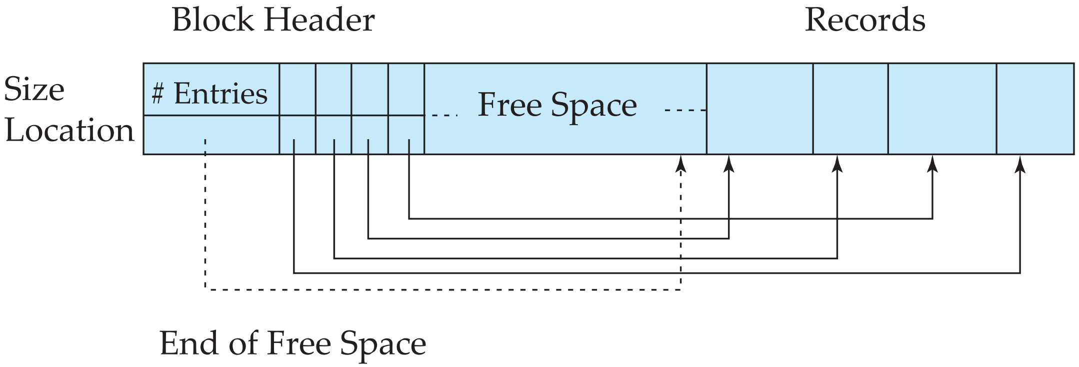
# Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Storage of multiple record types in a file.



- Store several records in blocks (slotted pages)
  - A record cannot be bigger than a slotted page!
  - This limits the size of records in a database, which is usually the case (at least, by default)
    - There are special types for big records, that are treated differently (remember the **clob**s and **blob**s in Oracle?)

# Variable-Length Records: Slotted Page Structure



- **Slotted page** are usually the size of a block
- 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.
- (Other) pointers should not point directly to record — instead, they should point to the entry for the record in header.

# Storing Large Objects

- E.g., blob/clob types
- Records must be smaller than pages
- Alternatives:
  - Store as files in file systems
  - Store as files managed by database
  - Break into pieces and store multiple tuples in separate relation
    - PostgreSQL TOAST

# Organization of Records in Files

- **Heap** – record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O
- **B$^+$-tree file organization**
  - Ordered storage even with inserts/deletes
  - More on this next week
- **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed
  - More on this in 2 weeks

# Heap File Organisation

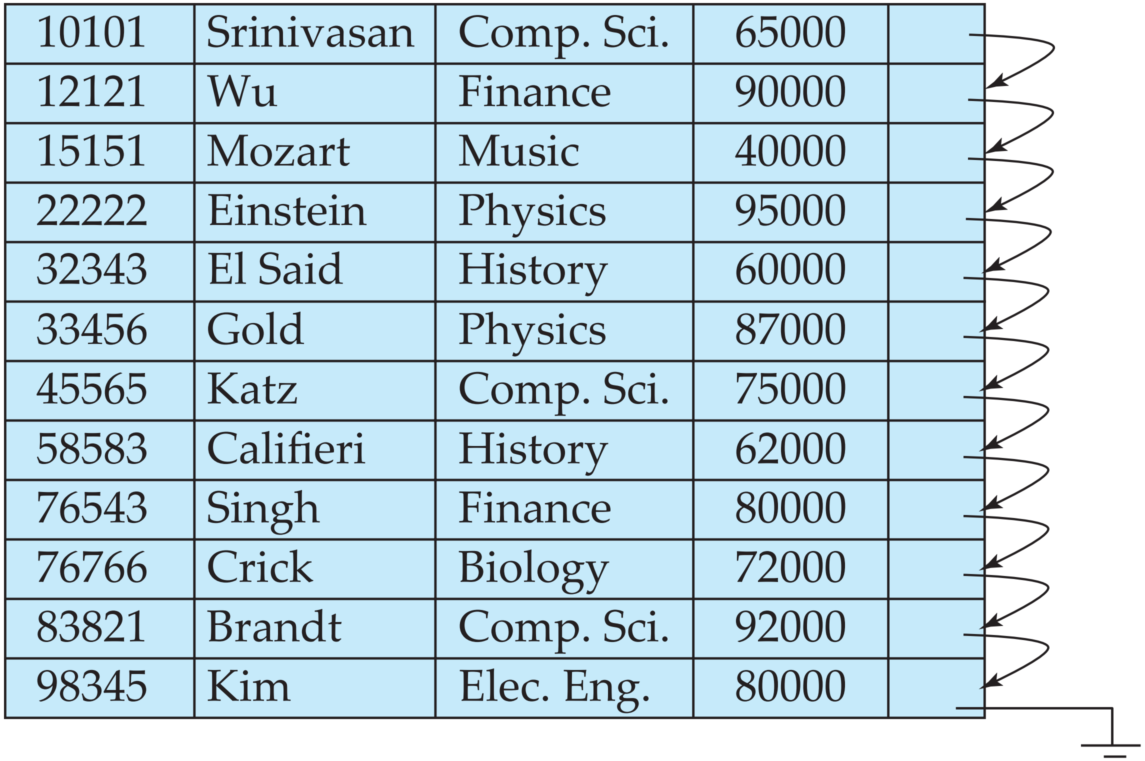
- Records can be placed anywhere in the file where there is free space
- Records usually do not move once allocated
  - Important to be able to efficiently find free space within file
- **Free-space map**
  - Array with 1 entry per block.  Each entry is a few bits to a byte, and records fraction of block that is free
  - In example below, 3 bits per block, value divided by 8 indicates fraction of block that is free

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 4 | 7 | 3 | 6 | 5 | 1 | 2 | 0 | 1 | 1 | 0 | 5 | 6 |

- Free space map written to disk periodically, OK to have wrong (old) values for some entries (will be detected and fixed)

# Sequential File Organization

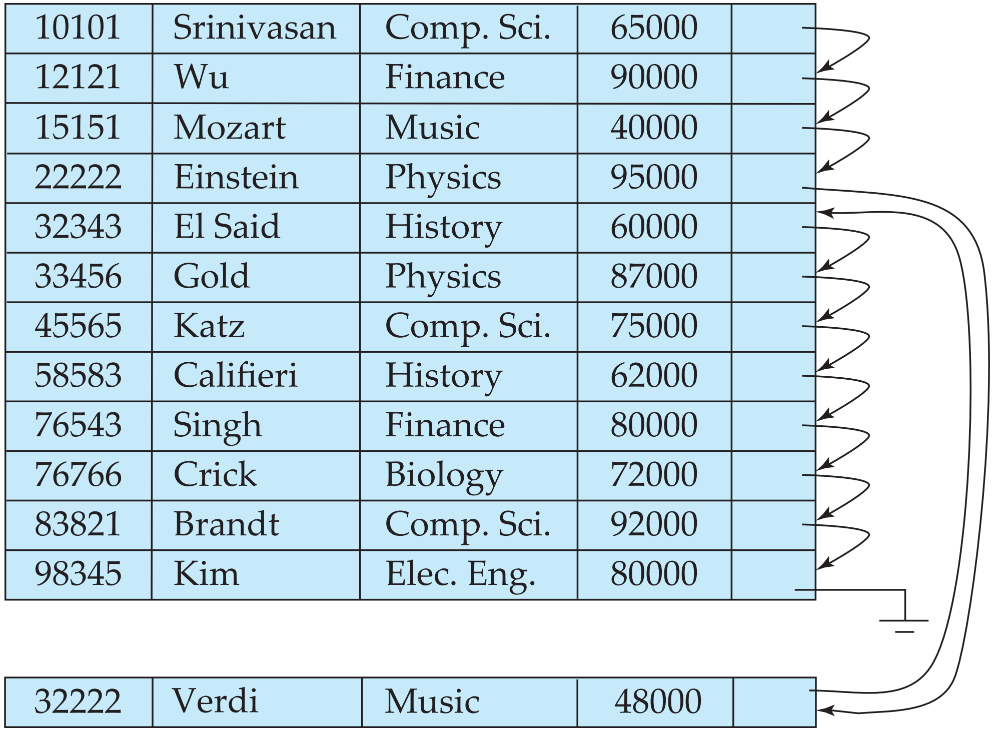
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key



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



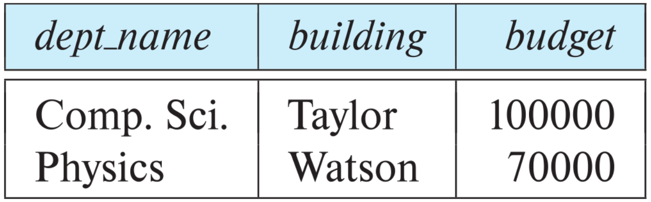
| | | | | |
|---|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 | |
| 12121 | Wu | Finance | 90000 | |
| 15151 | Mozart | Music | 40000 | |
| 22222 | Einstein | Physics | 95000 | |
| 32343 | El Said | History | 60000 | |
| 33456 | Gold | Physics | 87000 | |
| 45565 | Katz | Comp. Sci. | 75000 | |
| 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 | |

# Multitable Clustering File Organization

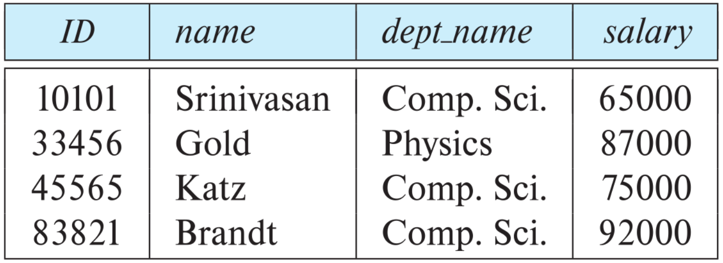
Store several relations in one file using a **multitable clustering** file organization

*department*



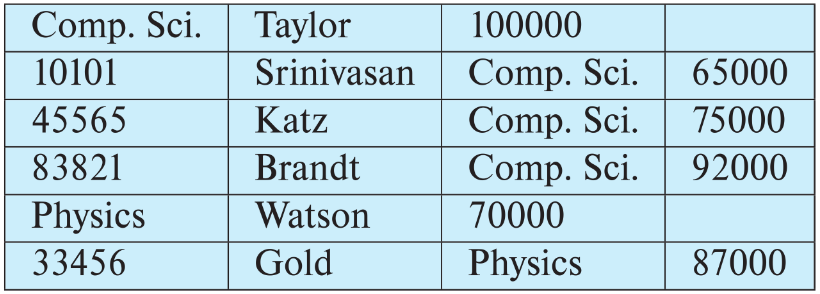
| *dept_name* | *building* | *budget* |
|---|---|---|
| Comp. Sci. | Taylor | 100000 |
| Physics | Watson | 70000 |

*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 |

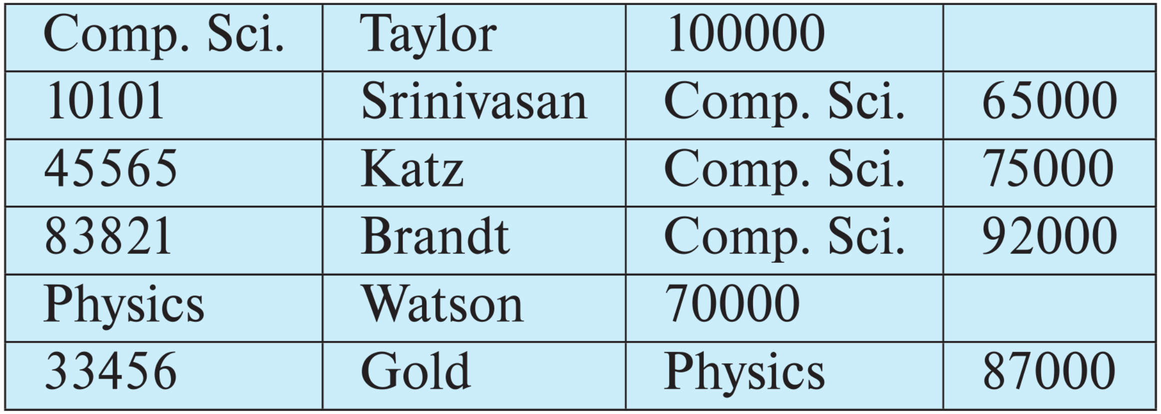
multitable clustering of *department* and *instructor*



| | | | |
|---|---|---|---|
| Comp. Sci. | Taylor | 100000 | |
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| Physics | Watson | 70000 | |
| 33456 | Gold | Physics | 87000 |

# Multitable Clustering File Organization (cont.)

- Good for queries involving *department* $\bowtie$ *instructor*, and for queries involving one single department and its instructors
- Bad for queries involving only *department*
  - But one can add pointer chains to link records of a particular relation
- Results in variable size records



| | | | |
|---|---|---|---|
| Comp. Sci. | Taylor | 100000 | |
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| Physics | Watson | 70000 | |
| 33456 | Gold | Physics | 87000 |

# Partitioning

- **Table partitioning**: Records in a relation can be partitioned into smaller relations that are stored separately
- E.g., *transaction* relation may be partitioned into *transaction_2018, transaction_2019, etc.*
- Queries written on *transaction* must access records in all partitions
  - Unless query has a selection such as *year*=2019, in which case only one partition in needed
- Partitioning
  - Reduces costs of some operations such as free space management
  - Allows different partitions to be stored on different storage devices
    - E.g., *transaction* partition for current year on SSD, for older years on magnetic disk

# File System

- In sequential file organisation, each relation (or partition of a relation) is stored in a file
  - One may rely in the file system of the underlying operating system

- Multitable clustering may have significant gains in efficiency
  - But this may not be compatible with the file system of the operating system

- Several large scale database management systems do not rely directly on the underlying operating system
  - The relations are all stored in a single (multitable) file
  - The database management system manages the file by itself
  - This requires the implementation of an own file system inside the DBMS

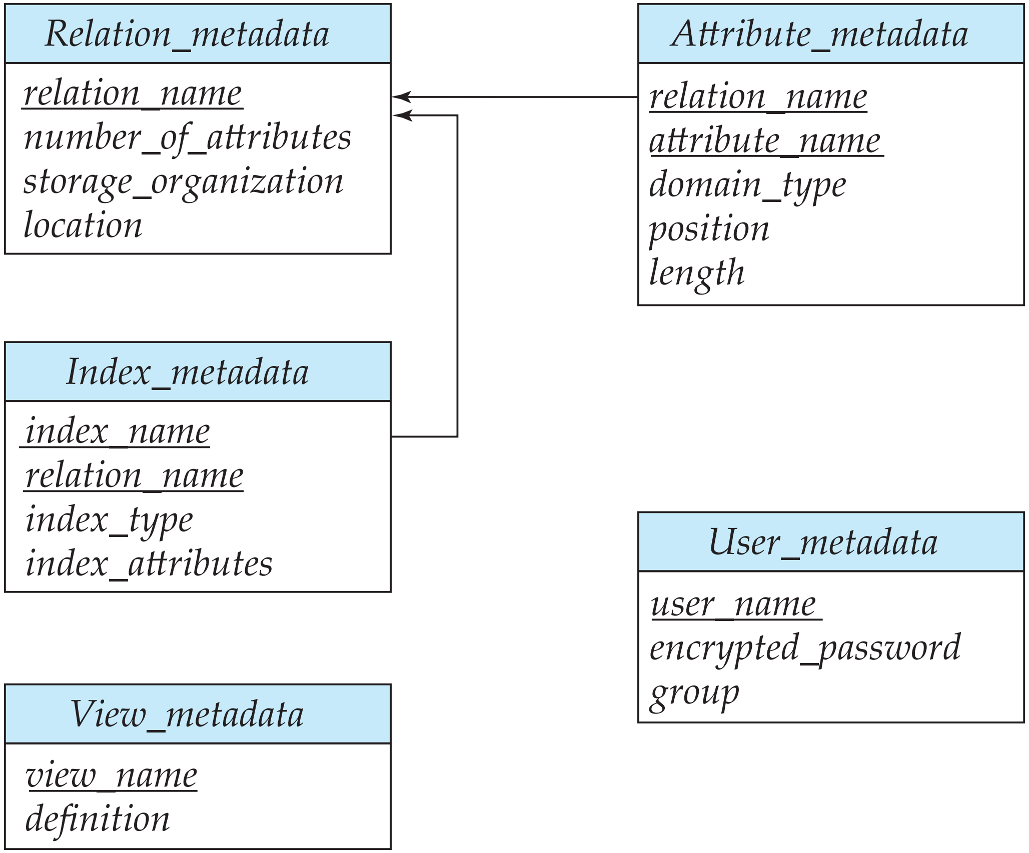
# Data Dictionary Storage

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 (more on this later)

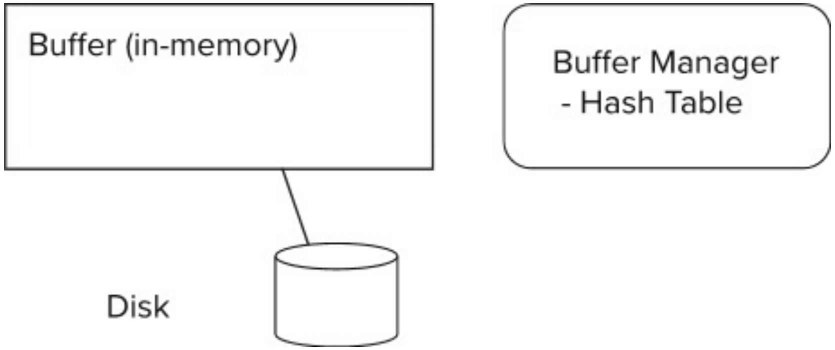
# Relational Representation of System Metadata

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



# Storage Access

- Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



# Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  - If the block is already in the buffer, buffer manager returns the address of the block in main memory
  - If the block is not in the buffer, the buffer manager
    - Allocates space in the buffer for the block
      - Replacing (throwing out) some other block, if required, to make space for the new block.
      - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
    - Reads the block from the disk to the buffer and returns the address of the block in main memory to requester.

# Buffer Manager

- **Buffer replacement strategy** (details coming up!)
- **Pinned block:** memory block that is not allowed to be written back to disk
  - **Pin** done before reading/writing data from a block
  - **Unpin** done when read /write is complete
  - Multiple concurrent pin/unpin operations possible
    - Keep a pin count, buffer block can be evicted only if pin count = 0

# Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
  - Idea behind LRU – use past pattern of block references as a predictor of future references
  - LRU can be bad for some queries
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- Example of bad access pattern for LRU: when computing the join of 2 relations r and s by a nested loop

```
for each tuple $tr$ of $r$ do
    for each tuple $ts$ of $s$ do
            if the tuples $tr$ and $ts$ match …
```

# Buffer-Replacement Policies (Cont.)

- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer

# File Organization in Oracle

- Oracle has its own buffer management, with complex policies
  - Oracle doesn’t rely on the underlying operating system’s file system
- A database in Oracle consists of **tablespaces**:
  - System tablespace: contains catalog meta-data
  - User data tablespaces
- The space in a tablespace is divided into **segments**:
  - Data segment
  - Index segment
  - Temporary segment (for sort operations)
  - Rollback segment (for processing transactions)
- Segments are divided into **extents**, each extent being a set of contiguous **database blocks**.
  - A database block need not be the same size of an operating system block, but is always a multiple

# File Organization in Oracle

- A standard table is organised in a heap (no sequence is imposed)
- Partitioning of tables is possible for optimisation
  - Range partitioning (e.g by dates)
  - Hash partitioning
  - Composite partitioning
- Table data in Oracle can also be (multitable) clustered (with **create cluster**)
  - One may tune the clusters to significantly improve the efficiency of query to frequently used joins.
- Hash file organisation (to be studied later) is also possible for fetching the appropriate cluster

- A database can be tuned by an appropriate choice for the organisation of data:
  - Choosing partitions
  - Appropriate choice of clusters
  - Hash or sequential
- Tuning makes the difference in big (real) databases!

# Chapter 14: Indexing

Sistemas de Bases de Dados 2019/20

Capítulo refere-se a: Database System Concepts, 7th Ed

# Outline

- Basic Concepts
- Ordered Indices
- B$^+$-Tree Index Files
- B-Tree Index Files
- Hashing
- Write-optimized indices
- Spatio-Temporal Indexing

# Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **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

| | |
|---|---|
| search-key | pointer |

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

# Index Evaluation Metrics

- Access time
- Insertion time
- Deletion time
- Space overhead

- 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

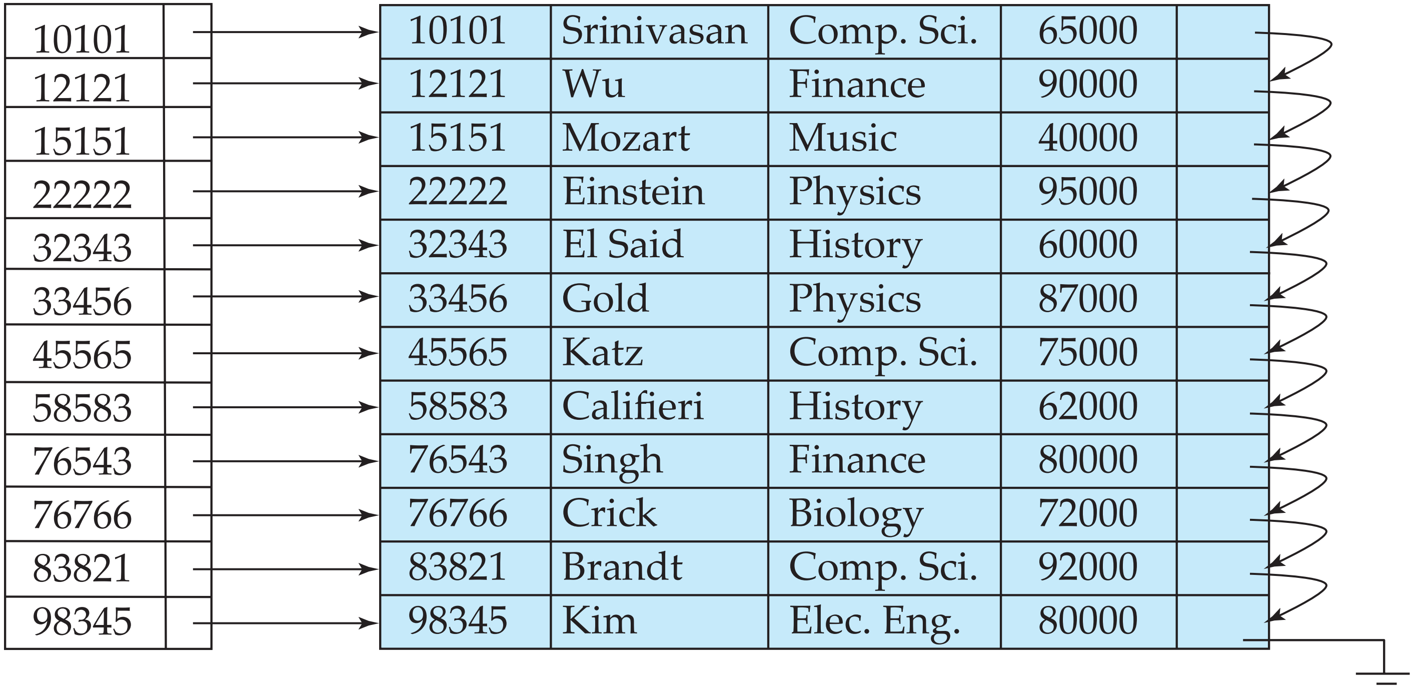
- The desired/usual access type strongly influences the choice of index

# Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **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

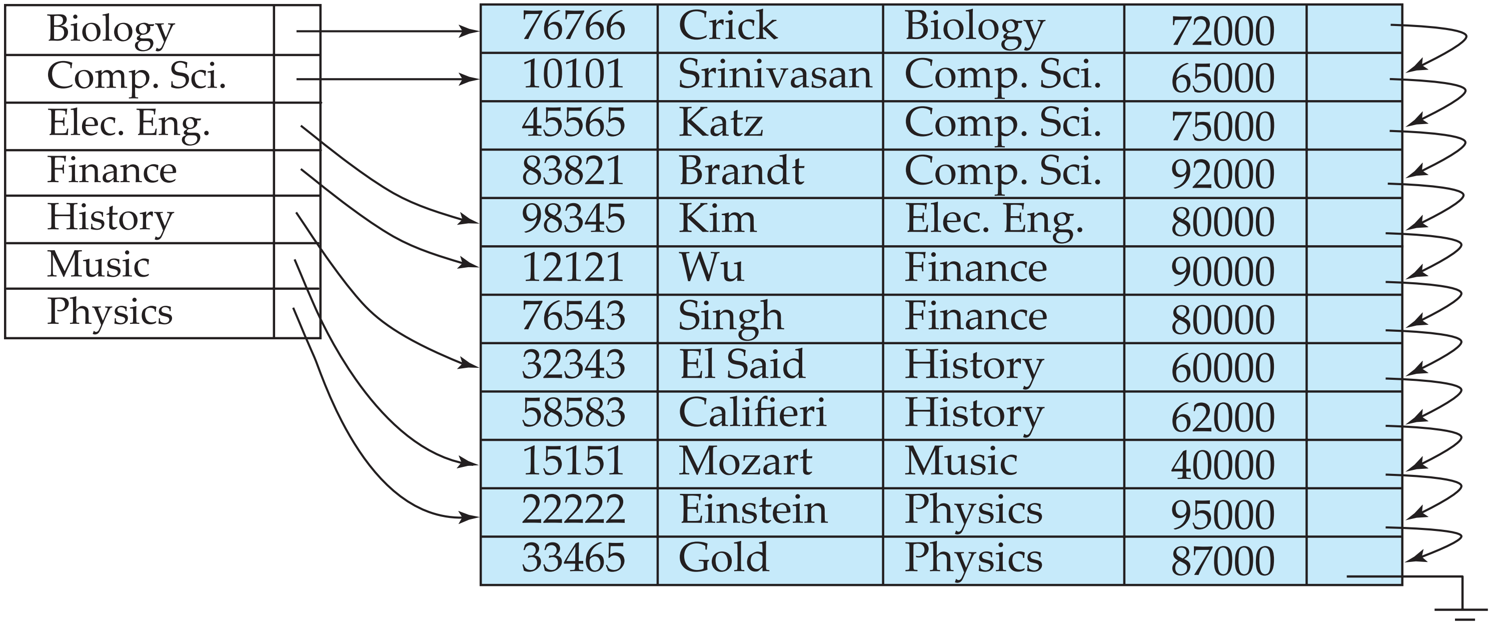


| | |
|---|---|
| 10101 | |
| 12121 | |
| 15151 | |
| 22222 | |
| 32343 | |
| 33456 | |
| 45565 | |
| 58583 | |
| 76543 | |
| 76766 | |
| 83821 | |
| 98345 | |

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

# Dense Index Files (Cont.)

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

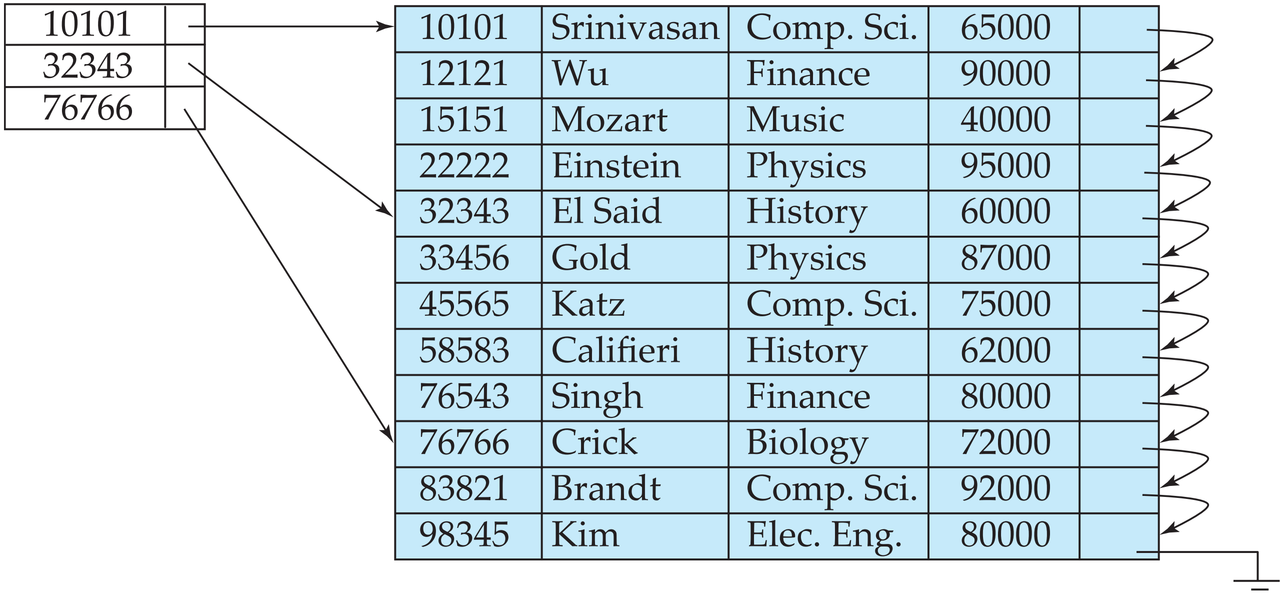


| | |
|---|---|
| Biology | |
| Comp. Sci. | |
| Elec. Eng. | |
| Finance | |
| History | |
| Music | |
| Physics | |

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

# Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - **Only** applicable in primary index, 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

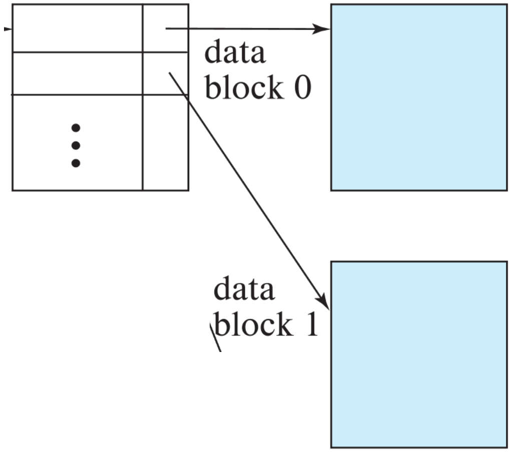


| | |
|---|---|
| 10101 | |
| 32343 | |
| 76766 | |

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

# 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**:
  - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

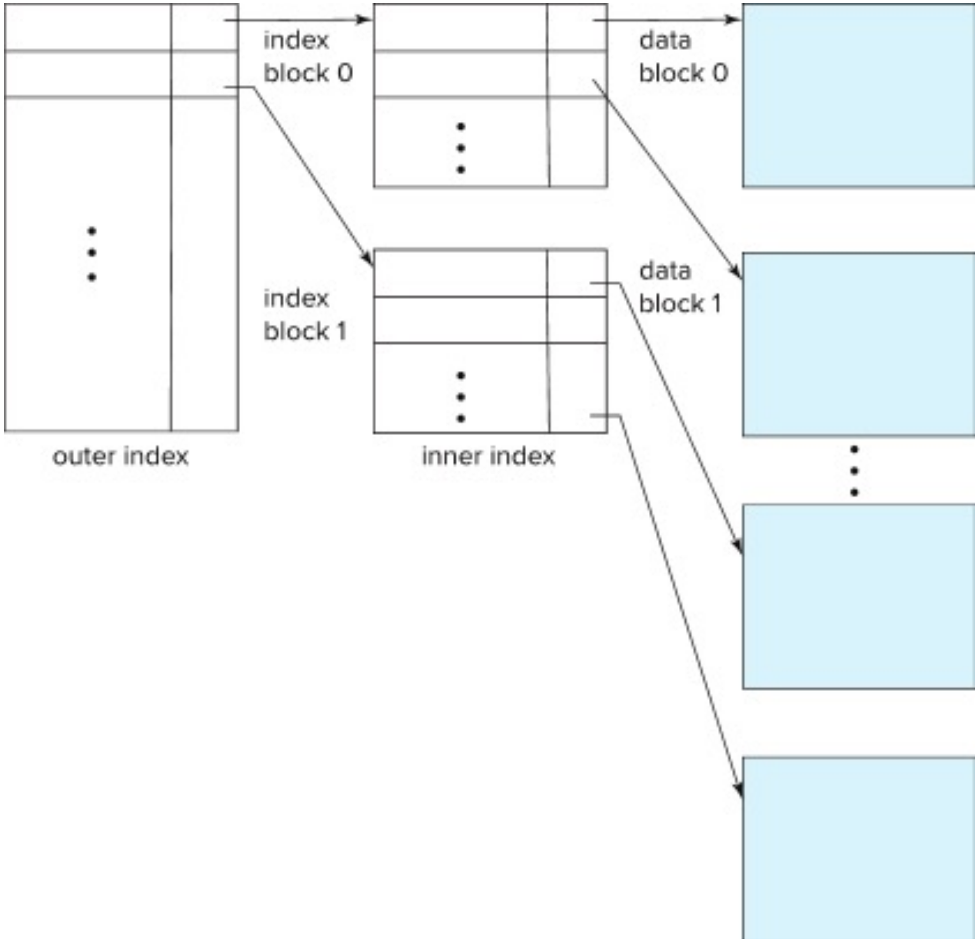


  - For unclustered index: sparse index on top of dense index (multilevel index)

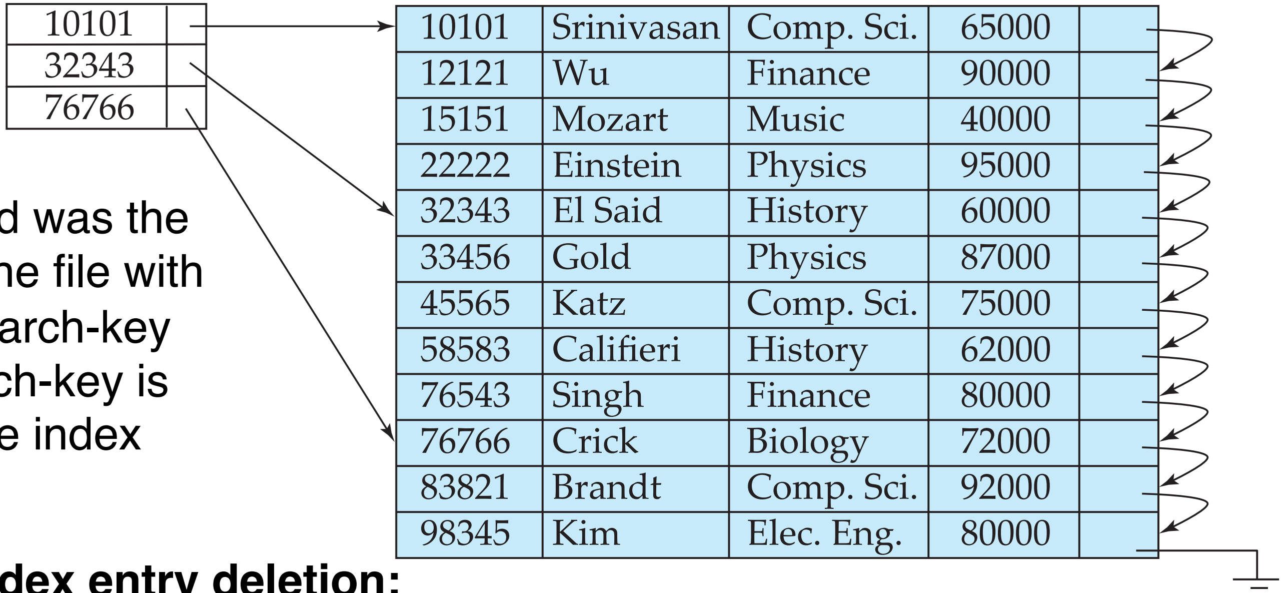
# Multilevel Index

- If the 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.

# Multilevel Index (Cont.)



# Index Update: Deletion



|  |
|---|
| 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 |

- 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

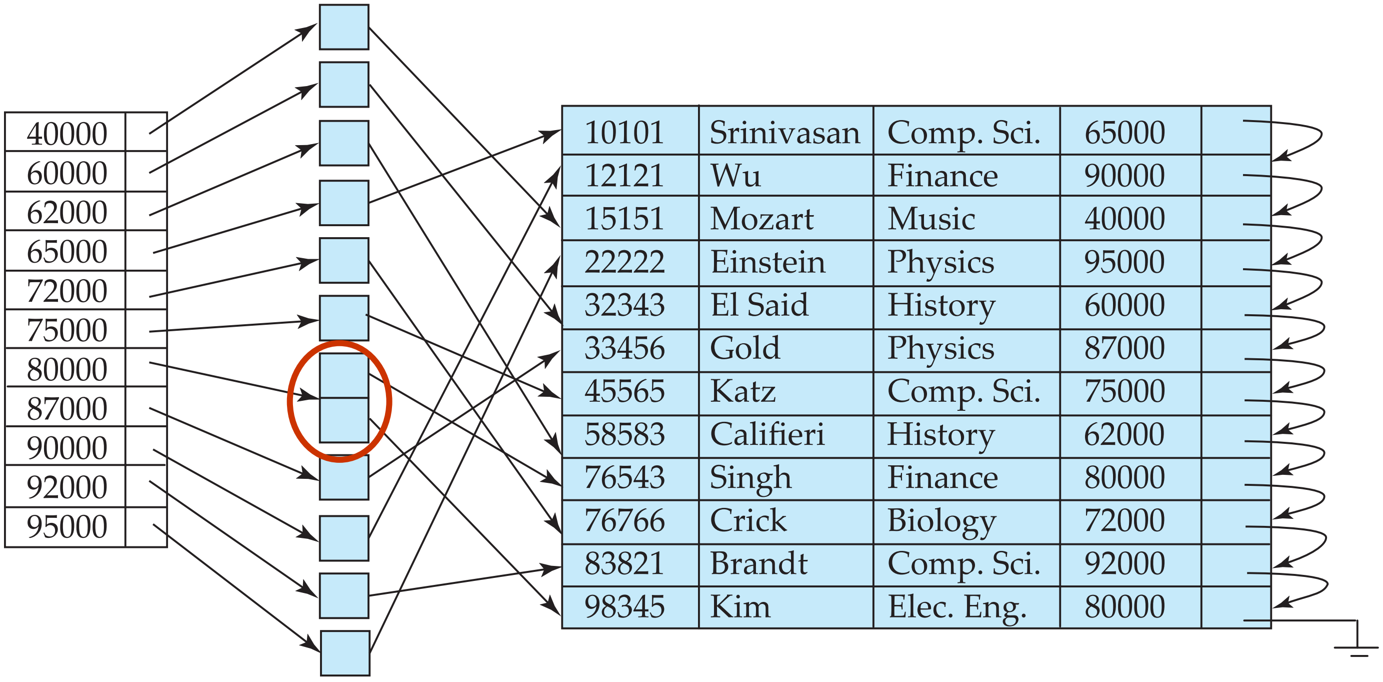
- Single-level index insertion:
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs 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 (as well as deletion) algorithms are simple extensions of the single-level algorithms
  - the outer indices are sparse, whereas the inner one is dense

# Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - *Example 1*: In the instructor relation stored sequentially by instructor ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or range of salaries
- We can have a secondary index with an index record for each search-key value

# Secondary Indices Example

- Secondary index on salary field of instructor



| |
|---|
| 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 |

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

# Primary vs Secondary 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