



Chapter 11: File Organization and and Indexing

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



File Organization

- The database is stored as a collection of operating systems **files**
 - Each file is a sequence of **records**
 - A record is a sequence of **fields**
- From the O/S point of view, each file consists of fixed-length **blocks** (also called **physical records**)
 - Blocks are the units of both storage allocation and data transfer
- One approach
 - Assume record size is fixed
 - Each file has records of one particular type only
 - Different files are used for different relations



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

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 : different alternatives:
 - ***move records $i + 1, \dots, n$ to $i, \dots, n - 1$***
 - 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 : different alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - **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

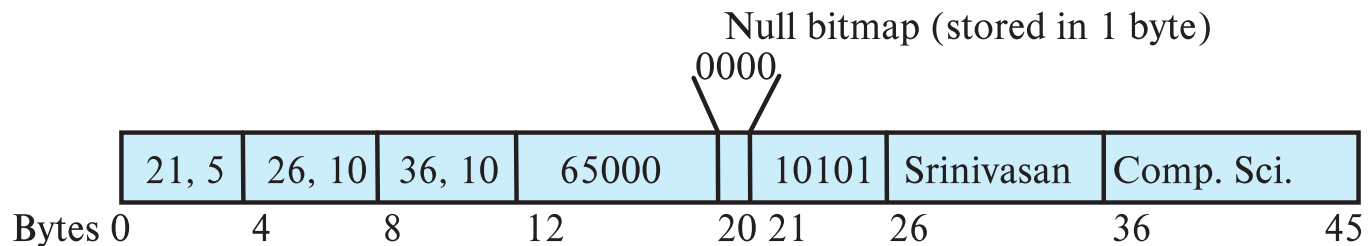
- Deletion of record i : different alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - move record n to i
 - **do not move records, but link all free records on a *free list***

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:
 - 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 (used in unnormalized data schemas)
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap





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
- **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed



Heap File Organization

- 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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Can have second-level free-space map
- In example below, each entry stores maximum from 4 entries of first-level free-space map

4	7	2	6
---	---	---	---

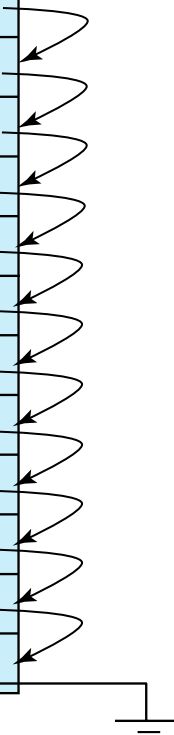
- Free space map is stored in a file, whose blocks are fetched into memory as required



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 and Binary Search

- For a sequential search for N items, the mean number of comparisons $E(X)$ is $\approx N/2$
 - Assuming each item has the same probability of being the required one (i.e., $1/N$), then the first item has probability $1/N$ of being the required item, the second item has probability $1/N$ of being the required item, and so on, and the search will terminate upon locating the required item:

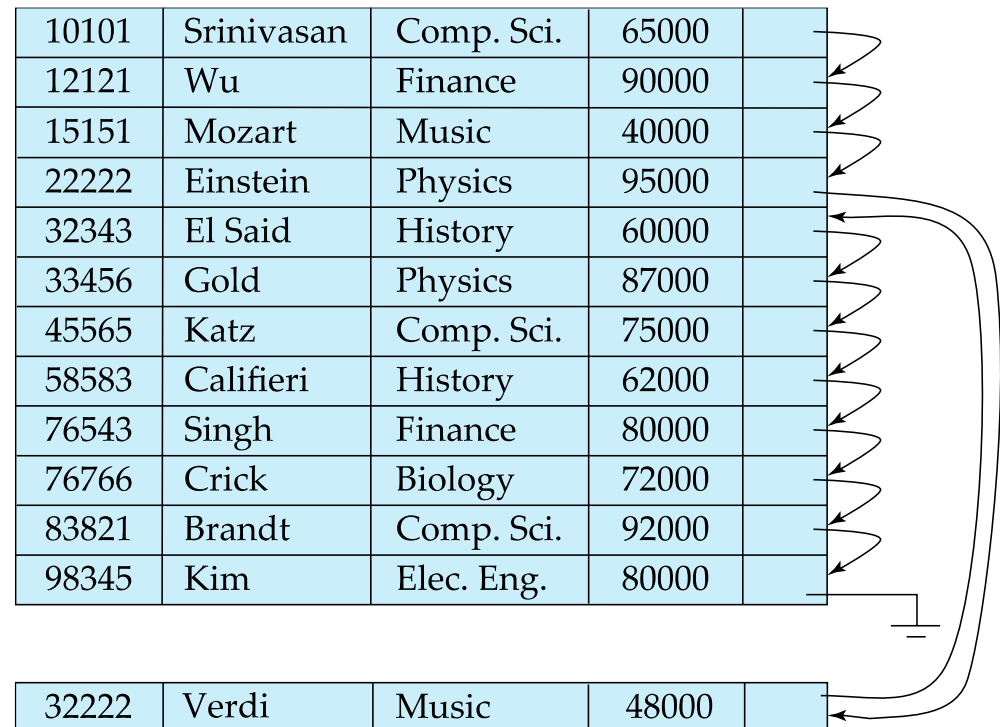
$$\begin{aligned} E(X) &= \frac{1}{N} + \frac{2}{N} + \frac{3}{N} + \cdots + \frac{N}{N} \\ &= \frac{1}{N} \sum_{k=1}^N k = \frac{1}{N} \times \frac{N(N+1)}{2} = \frac{(N+1)}{2} \approx \frac{N}{2} \end{aligned}$$

- Binary Search for N ordered items, the maximum number of comparisons k is $\approx \log_2 N$
 - Consider a line of length N , and keep dividing the length into two until only one item remains
 - $\frac{N}{2^k} = 1 \Rightarrow N = 2^k \Rightarrow k = \log_2 N$



Sequential File Organization

- 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





Multitable Clustering File Organization

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

department

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

- good for queries involving *department* \bowtie *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



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 is 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



Data Dictionary Storage

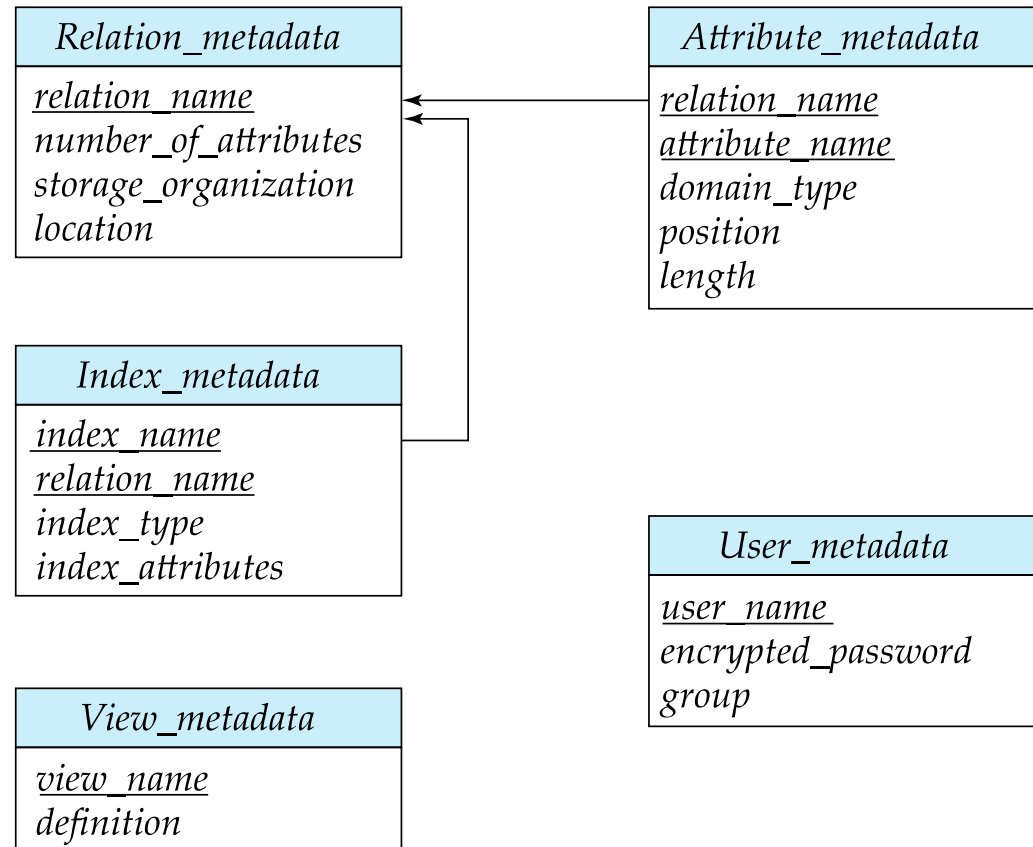
The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- 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



Relational Representation of System Metadata

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





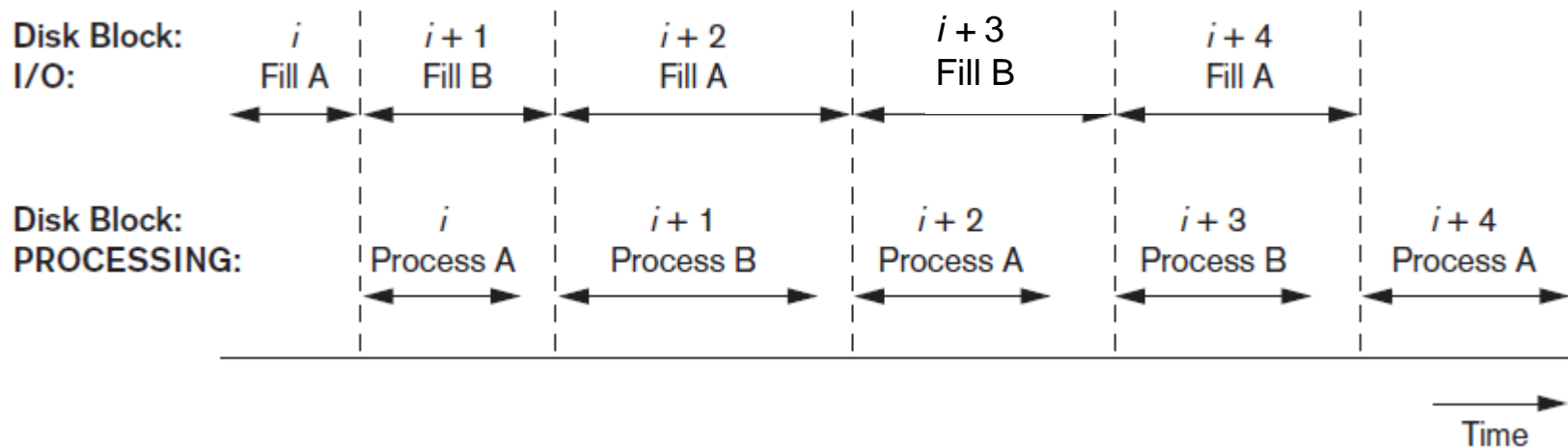
Storage Access

- 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



Buffering of Blocks

- Double buffering can be used to read continuous stream of blocks
- Use of two buffers, A and B, for reading from disk
- More than 2 buffers can be used





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**
 - e.g., LRU, FIFO, MRU
- **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
- **Shared and exclusive locks on buffer**
 - Needed to prevent concurrent operations
 - **Locking rules:**
 - Only one process can get exclusive lock at a time
 - Shared lock cannot be concurrently held with exclusive lock
 - Multiple processes may be given shared lock concurrently



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
- 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



Buffer-Replacement Policies

- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **FIFO** strategy – first-in, first-out
- **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



Column-Oriented Storage

- Also known as **columnar representation**
- Store each attribute of a relation separately
- Example

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



Column-Oriented Representation

- Benefits:
 - Reduced I/O if only some attributes are accessed
 - Improved CPU cache performance
 - Improved compression
 - **Vector processing** on modern CPU architectures which allows a CPU operation to be applied in parallel on a number of elements of an array (e.g., comparing an attribute with a constant)
- Drawbacks
 - Cost of tuple reconstruction from columnar representation
 - Cost of tuple deletion and update
 - Cost of decompression
- Columnar representation often more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
 - Called **hybrid row/column stores**



Indexing Concepts

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Basic Concepts

- Indexing mechanisms is used to speed up access to desired data.
 - E.g., author catalog in library
- **Search Key** - attribute to a set of attributes used to look up records in a file.
- An **index is a file** consists of records (called **index entries**) of the form

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

- Index files are typically much smaller than the original file



Index Evaluation Metrics

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



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
 - 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
 - Key values may or may not be unique
- **Indexed-sequential file (ISAM – Indexed Sequential Access Method)**: sequential file ordered on a search key, with an index on the search key



Dense Index Files

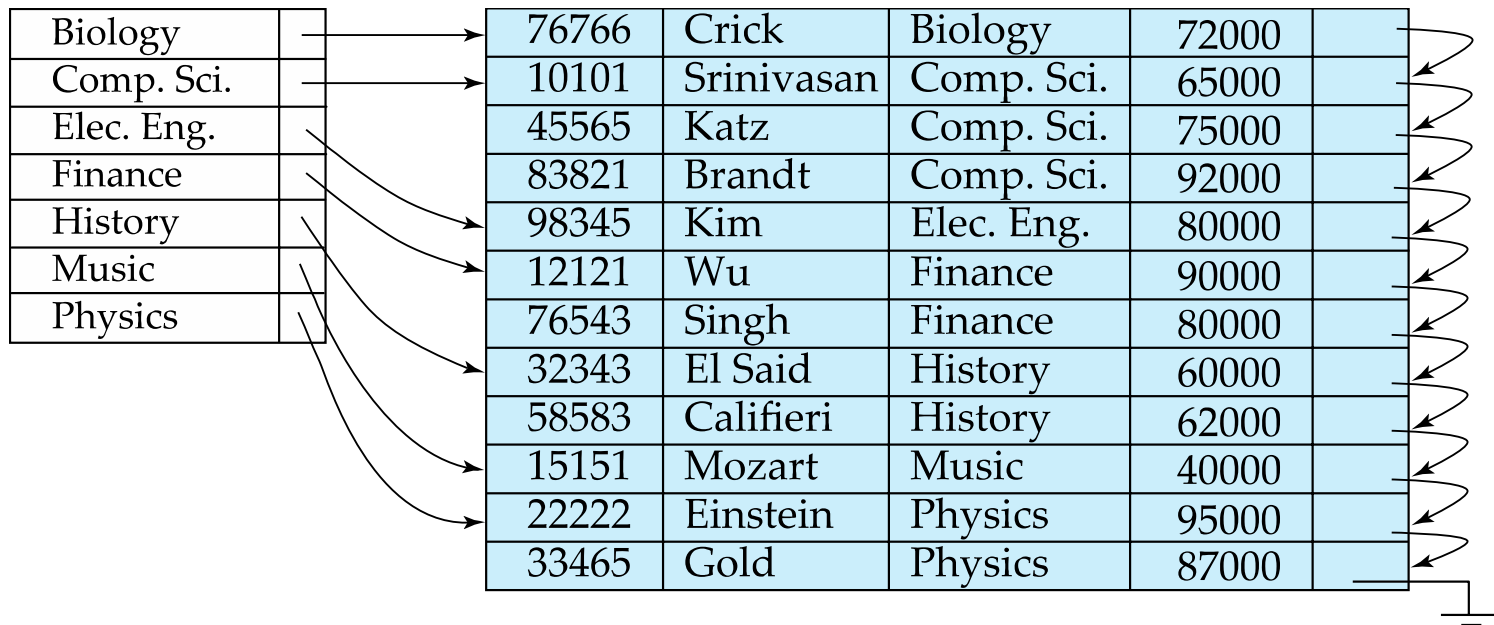
- **Dense index** — Index record appears for every search-key value in the file
- E.g. index on *ID* attribute of *instructor* relation

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



Dense Index Files

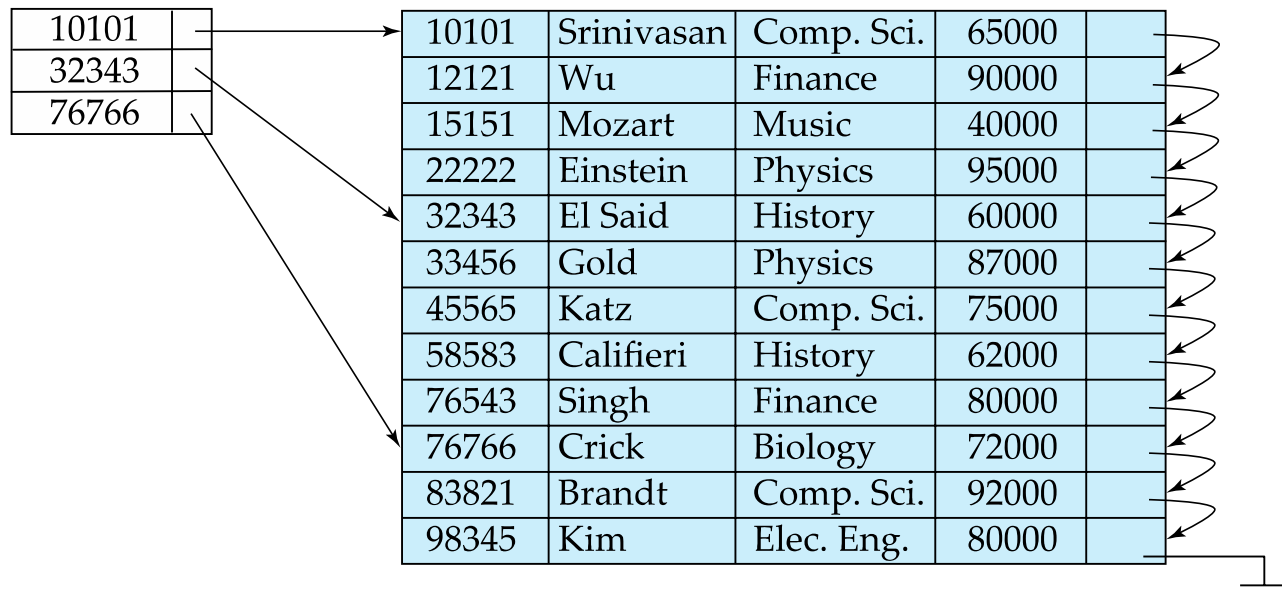
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*





Sparse Index Files

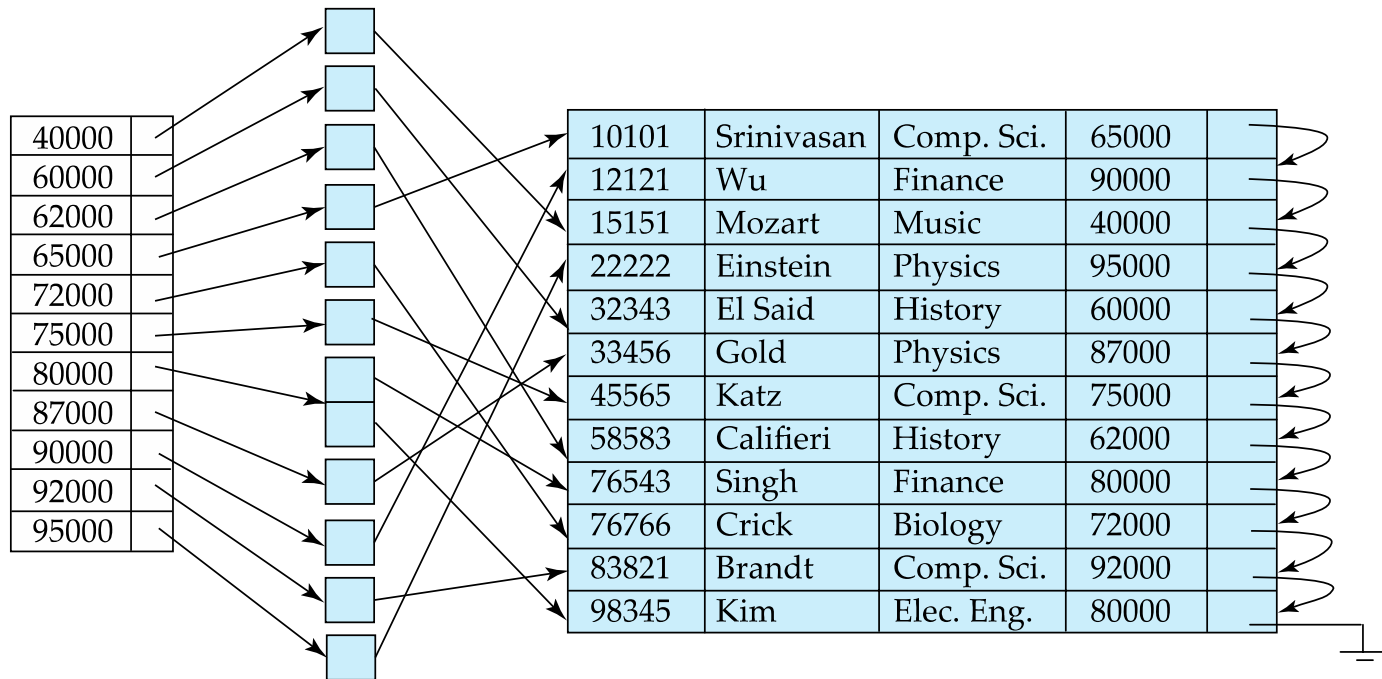
- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $\leq K$
 - Search file sequentially starting at the record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





Secondary Index

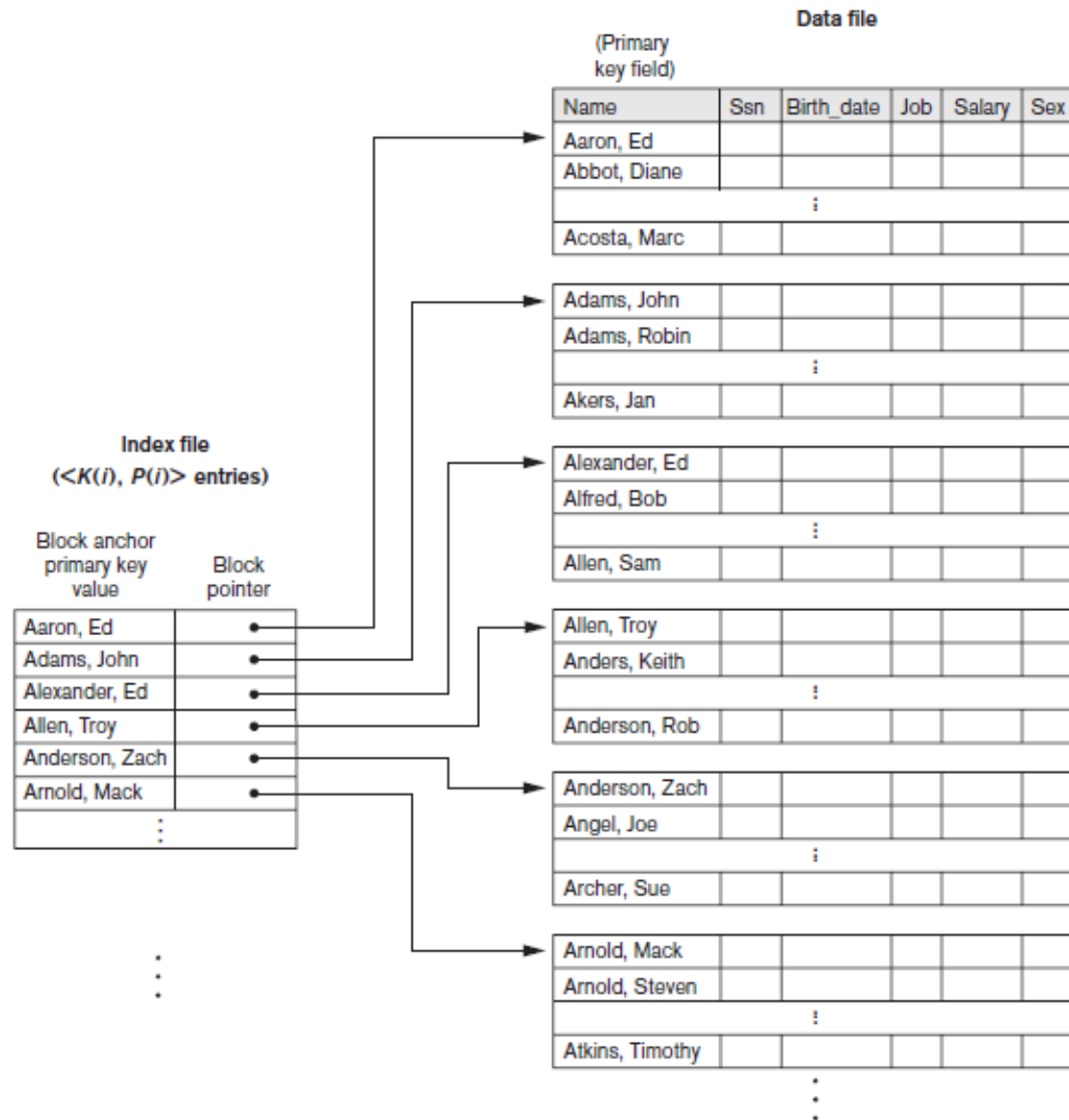
- Secondary index on salary field of instructor



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



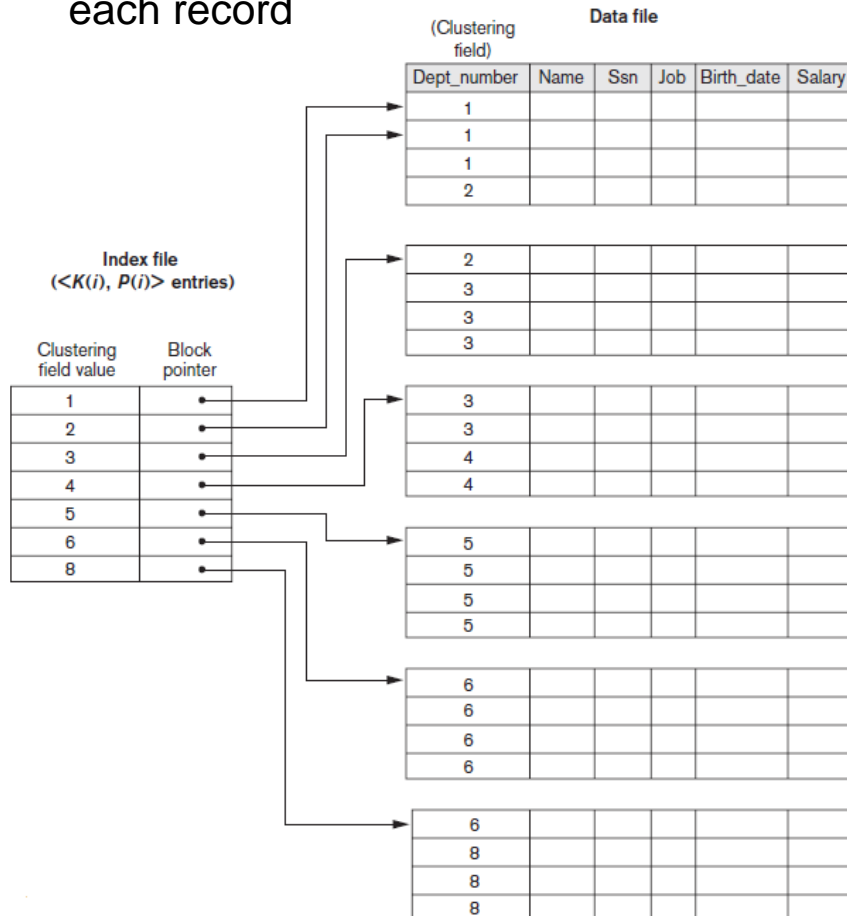
Primary Index





Clustering Indexes

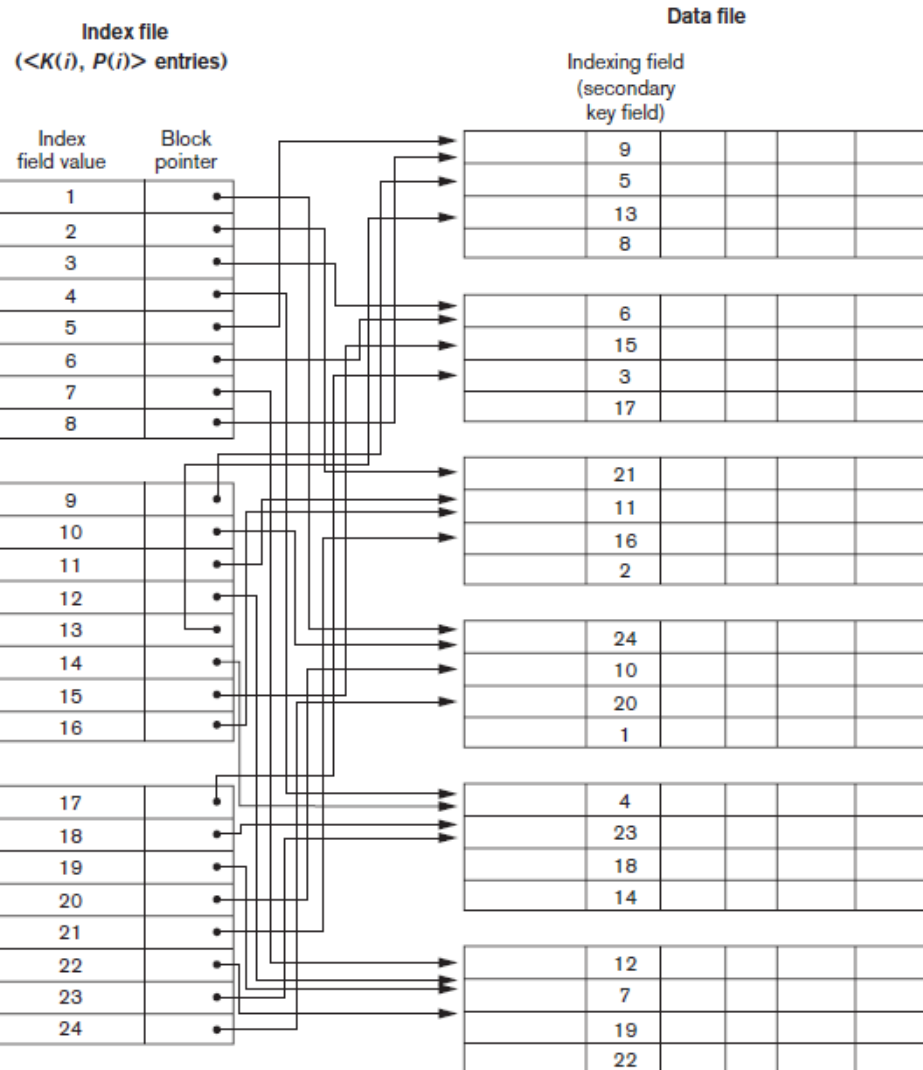
- Clustering field
 - records are physically ordered on a nonkey field without a distinct value for each record



A clustering index on the Dept_number which is an ordering nonkey field of an Employee file



Secondary Index



Dense secondary index (with block pointers) on a nonordering key field of a file.



Multilevel 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



A two-level ISAM (indexed sequential access method) organization

