

# **File Organization and Indexing**

# File Organization

- ❑ The database is stored as a collection of *files*.
- ❑ Each file is organized logically as a sequence of *records*.  
A record is a sequence of fields.
- ❑ These records are mapped into disk blocks.
- ❑ File is logically partitioned into fixed-length storage units called blocks, which are the units of both storage allocation and data transfer.
- ❑ Most databases use block sizes of 4 to 8 kilobytes by default

# Example-COMPANY DATABASE

## EMPLOYEE

EMP_ID	EMP_NAME	ADDRESS	DEP_ID
1	John	Delhi	14
2	Robert	Gujarat	12
3	David	Mumbai	15
4	Amelia	Meerut	11
5	Kristen	Noida	14
6	Jackson	Delhi	13
7	Amy	Bihar	10
8	Sonoo	UP	12

## DEPARTMENT

DEP_ID	DEP_NAME
10	Math
11	English
12	Java
13	Physics
14	Civil
15	Chemistry

# **File Organization(Cont'd)**

- One approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

# RECAP

- Consider a database University having the tables instructor, student, class and subject. The number of files required to store data in this database is

- Consider a table instructor(ins\_i,name,deptname,salary,designation) , each record occupies 48 bytes. The disk block size is 192 bytes. How many instructor records are stored in each block?

# Fixed-Length Records

## □ Simple approach:

- Store record  $i$  starting from byte  $n * (i)$ , 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  $i + 1, \dots, n$  to  $i, \dots, n - 1$
- move record  $n$  to  $i$
- do not move records, but link all free records on a *free list*

record 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

record 1

record 2

record 3

record 4

record 5

record 6

record 7

record 8

record 9

record 10

record 11

10101	Srinivasan	Comp. Sci.	65000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000



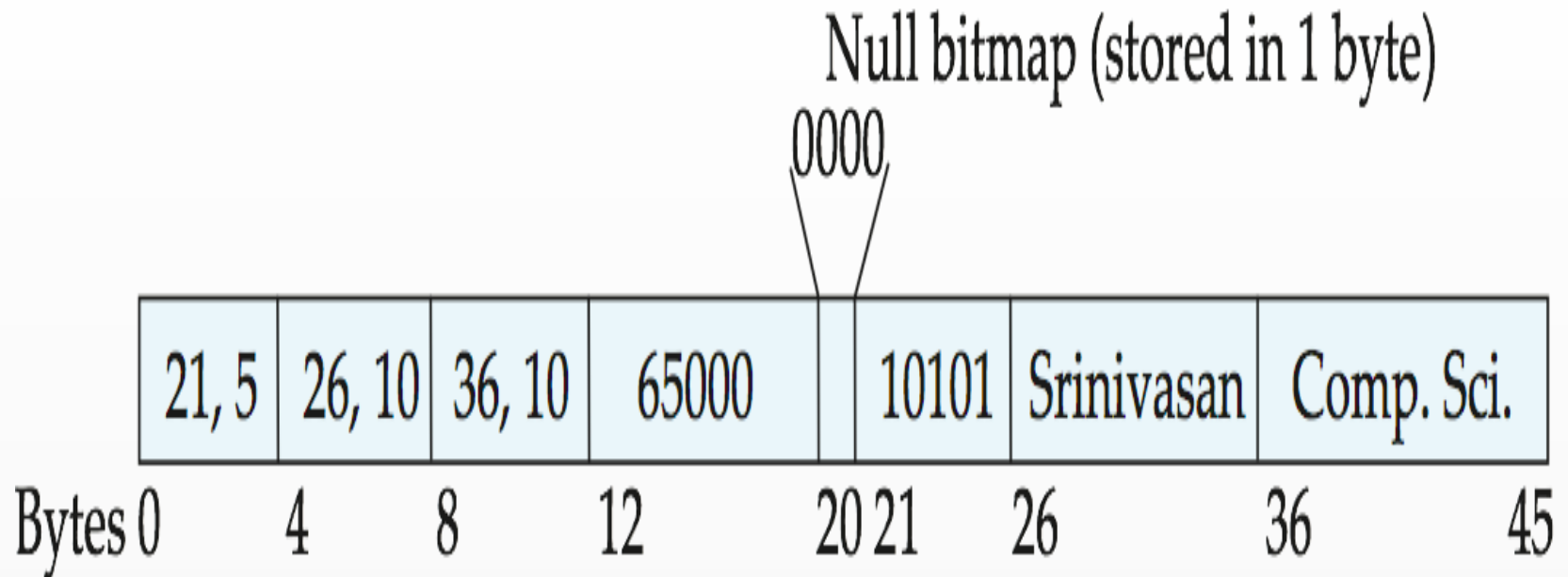
# Variable-Length Records

- 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 (arrays or multisets)
- Different techniques for implementing variable-length record must solve two different problems:
  - 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

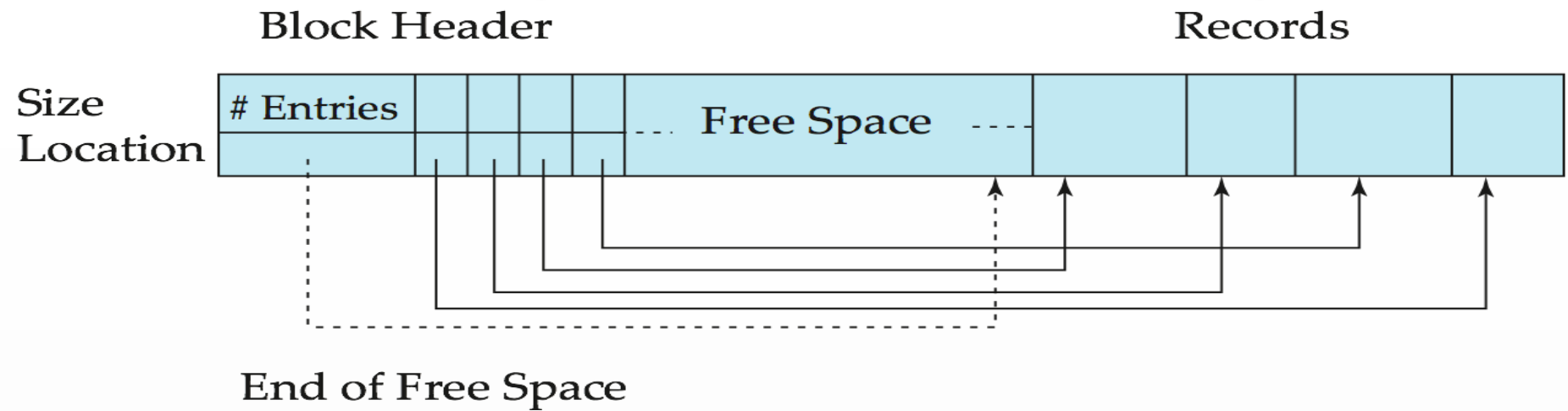
# Representation of records with variable length attributes

- Attributes are stored in order
- Representation of records consists of two parts: an initial part with fixed length attributes, followed by data for variable-length attributes
- Fixed-length attributes are allocated as many bytes as required to store their value.
- Variable-length attributes are represented by a pair(offset, length), where offset denotes where the data for that attribute begins within the record, and the length is the length in bytes of the variable sized attribute
- Null values represented by null-value bitmap, which indicates which attributes of the record have a null value.

# Variable-Length Records



# 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.

# Organization of Records in Files

- **Heap** – a 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
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. 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

# Example

## 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

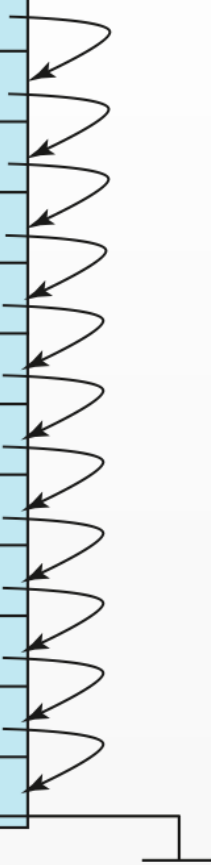
multitable clustering  
of *department* and  
*instructor*

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

# 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	

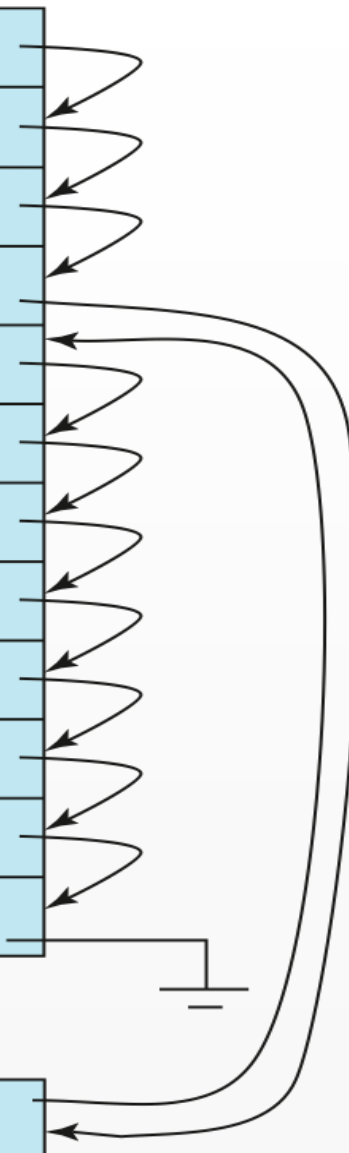




# Sequential File Organization (Cont.)

- ❑ 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

# Sequential File Organization (Cont.)

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		

# Basic Concepts

- **Indexing** mechanisms used to **speed up access** to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or 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 types** supported efficiently.

e.g.,

- records with a specified value in the attribute *where deptno='D1'*
- or records with an attribute value falling in a specified range of values.  
*where salary >=40000 and salary <=299999*

- **Access time**- to find a particular data item, or set of items
- **Insertion time**- find the correct place to insert the new data, to update the index structure
- **Deletion time** - takes to find the item to be deleted , to update the index structure.
- **Space overhead**-additional space occupied by an index structure
- An attribute or set of attributes used to look up records in a file is called a *search key* and it is **not necessarily same as primary key/candidate key**

# Ordered Indices

- In an **ordered index**, index entries in the **index file** are stored sorted on the **search key value**. E.g., author catalog in library.
- **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.
    - ▶ **There are two types under this**
      - **Dense index & Dense Clustering Index**
      - **Sparse Index**
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- **Index-sequential file**: ordered sequential file with a primary index.

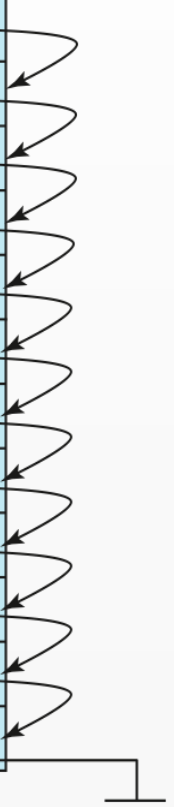
# Ordered Indices (Contd..)

- Index is created on ID attribute of Instructors which is in the sorted order
- Actual records are also in the order of ID
- These type files known as **Index-Sequential files**

**ID**

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	→


In fact this is dense index also-



# Dense Index Files

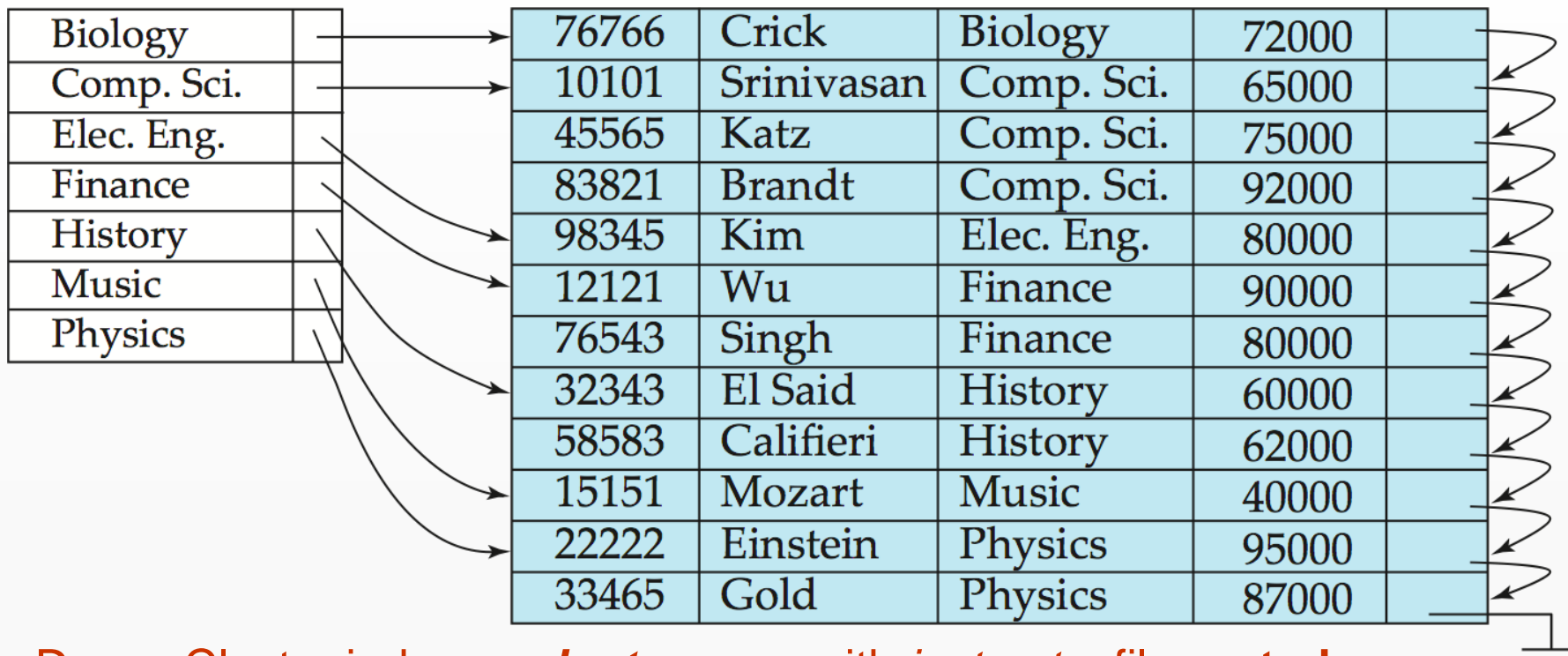
- **Dense index** — Index record appears for **every search-key value (for every ID value)** in the file.
- E.g. index on *ID* attribute of *instructor* relation

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 Cluster Index Files

- In a **dense clustering** index, the index record contains the **search-key value** and a **pointer to the first data 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.



Dense Cluster index on **dept\_name**, with *instructor* file **sorted on dept\_name**

In a **dense non-clustering index**, the index must store a **list of pointers to all records** with the same search-key value.



# Sparse Index Files

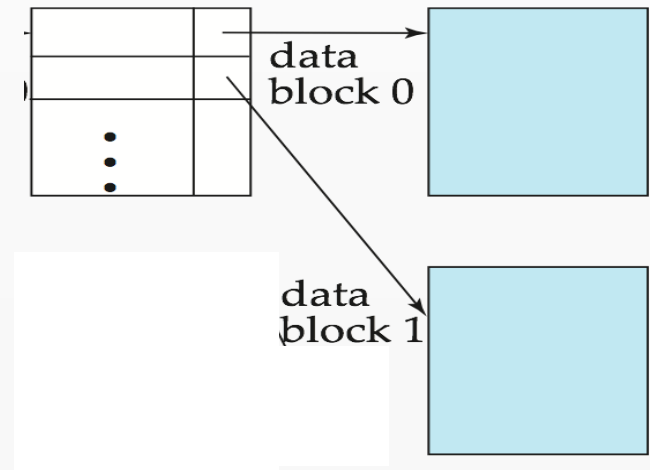
- ❑ **Sparse Index:** contains index records for **only some search-key values**.
  - ❑ **10101 search key** for set of records ranging search keys 10101 to 32342 and **32343 search key** for set of records ranging search keys 32343 to 76765 and so on.
  - ❑ Applicable when records are **sequentially 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

10101		10101	Srinivasan	Comp. Sci.	65000	
32343		12121	Wu	Finance	90000	
76766		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:** There is a trade-off that the system designer must make between access time and space overhead depending on application under consideration.
- ❑ A good compromise is to have a sparse index with **one index entry per block**.

Bringing one Block at a time from Hard disk  
(involves 1 seek & 1 Rotational Delay)  
to memory and searching required key in  
the block (Searching in memory less time)



# Index-Insert & Delete Operations

- ❑ Regardless of what form of index is used, every index must be updated whenever
  - ❑ A record is either **inserted** into or **deleted** from the file.
  - ❑ A record in the file is **updated**, any index whose search-key attribute is affected by the update must also be updated
- ❑ Such a record update can be modeled as
  - ❑ A deletion of the old record, followed by an insertion of the new value of the record, which results in an index deletion followed by an index insertion.
- ❑ As a result we only need to consider insertion and deletion on an index
- ❑ We first describe algorithms for updating single-level indices.

# Index Update: Insertion

**Insertion.** First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is **dense or sparse**:

- **Dense indices:**

1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.

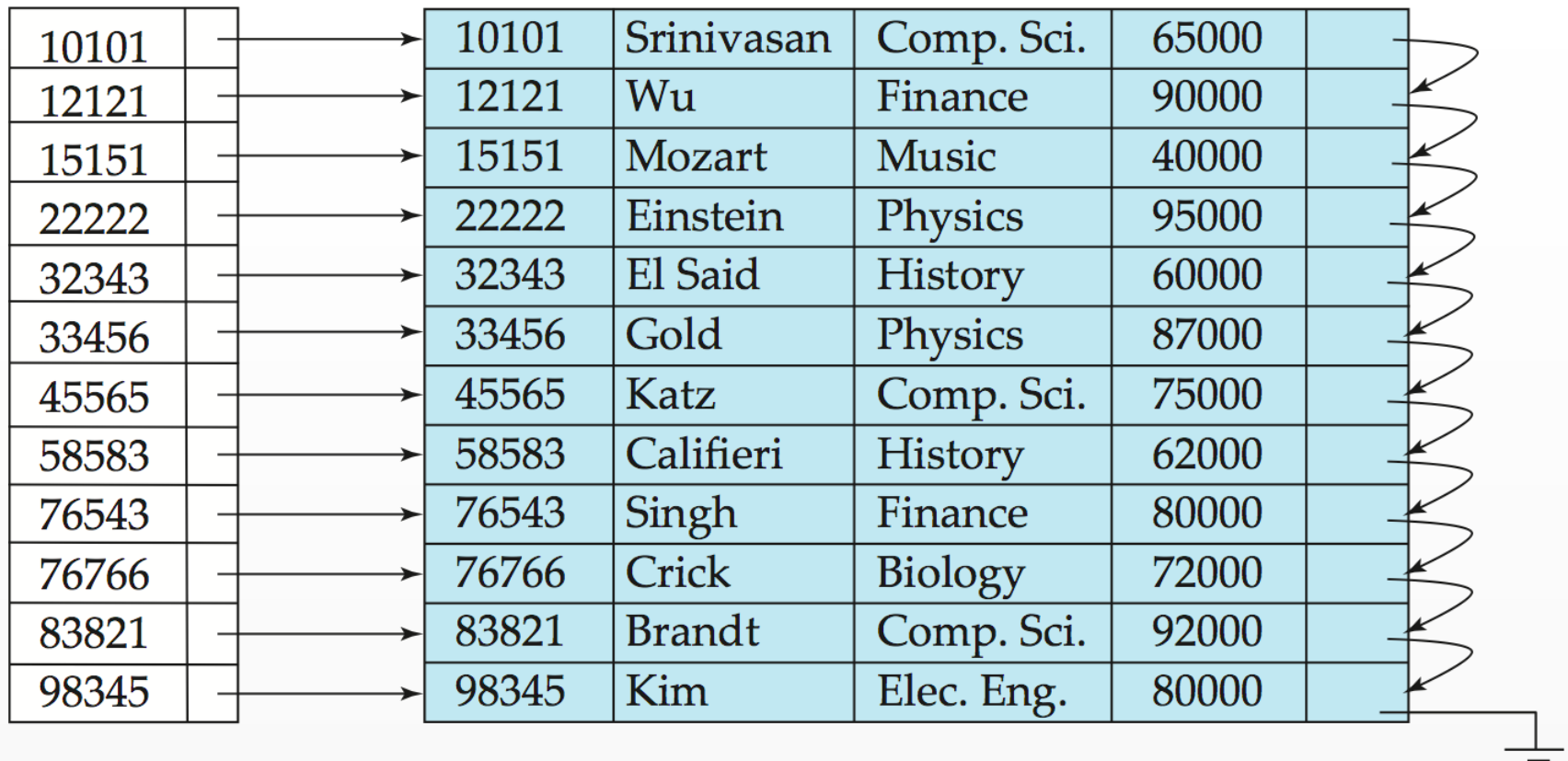
2. **Otherwise** the following actions are taken:

- a. **Dense & Non-Clustering-** If the **index entry stores pointers to all records** with the same search key value, the system adds a pointer to the new record in the index entry.

- b. **Dense & Clustering-** **Otherwise, the index entry stores a pointer to only the first record** with the search-key value. The system then places the record being inserted after the other records with the same search-key values.

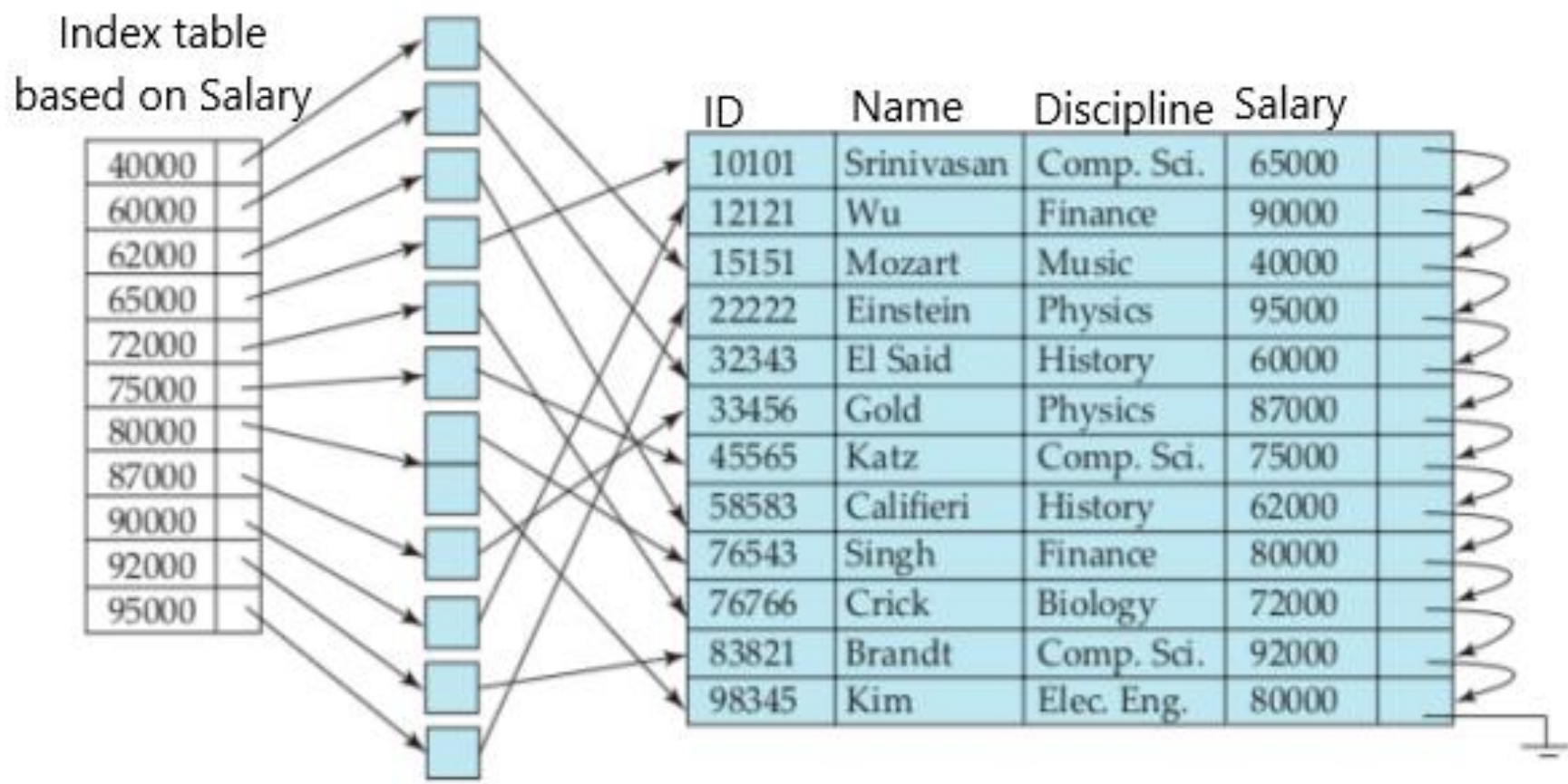
1. If the search-key value does not appear in the index...

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	



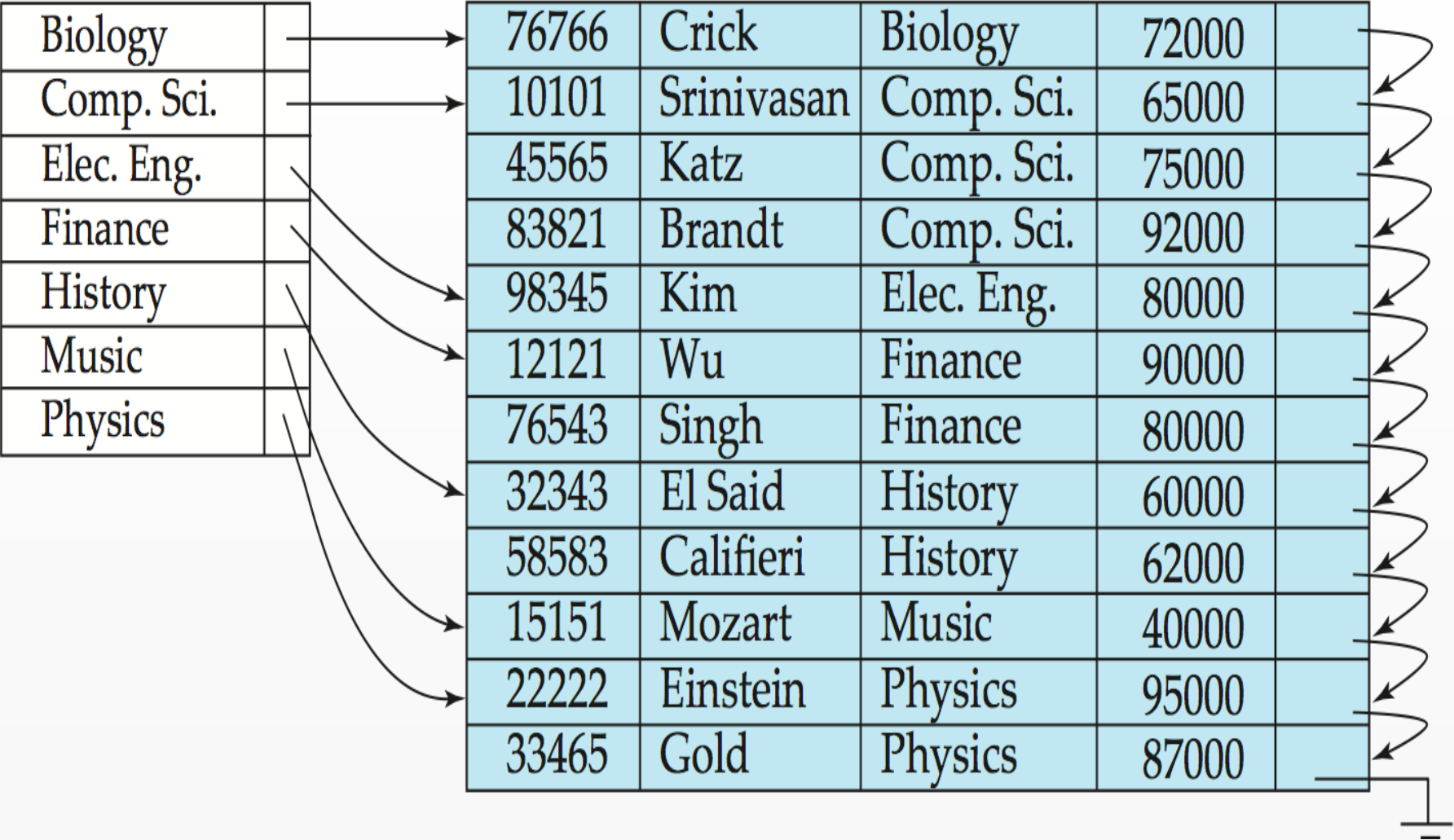
If inserting **<77777,XYZ,ABC,79000>** , as there is no entry corresponding to **77777** then insert the record in file and create an index entry corresponding to **77777** after 76766 and pointer pointing to **<77777,XYZ,ABC,79000>**

2a. If the index entry stores **pointers to all records (Dense non-clustering)**



If inserting the record <85555,John,Comp.Sc.,50000.>

2b. If the index entry stores a **pointer to only the first record with same search-key** value...





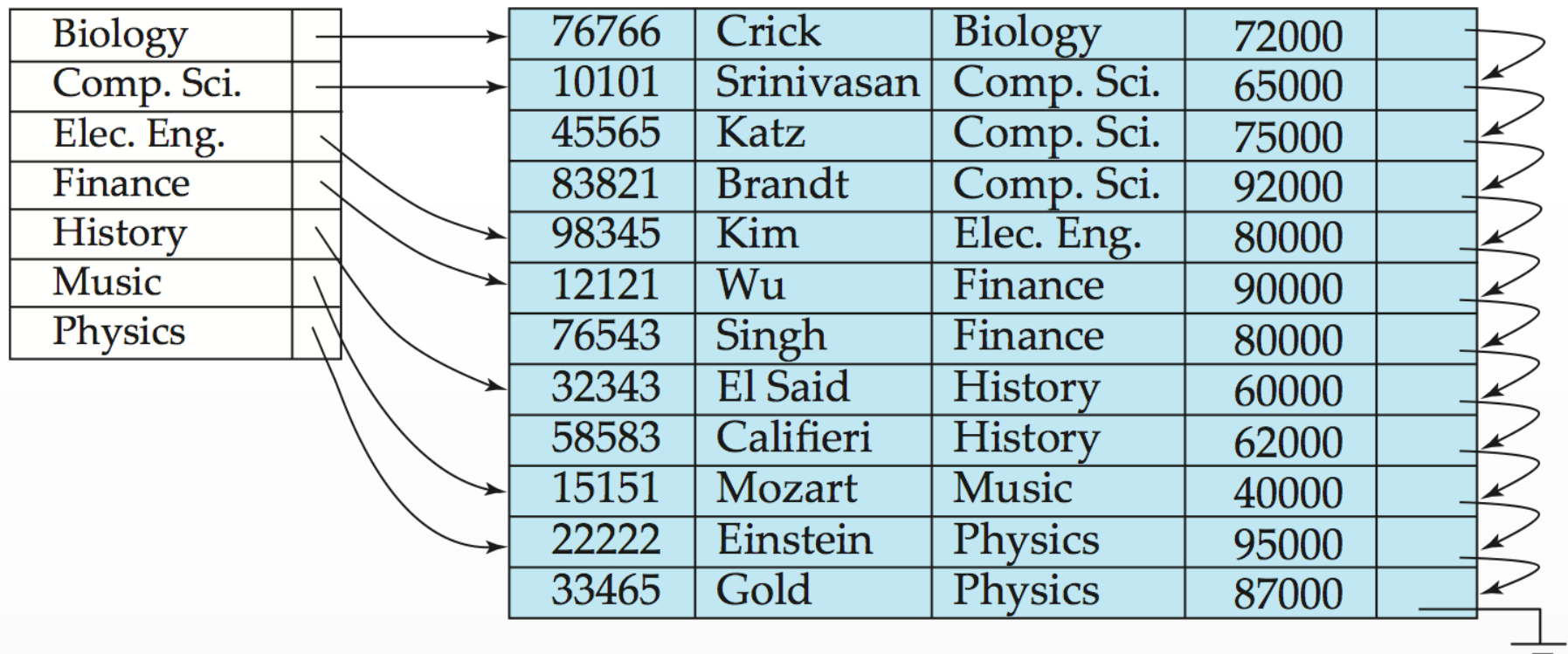
# Index Update: Deletion (if Dense)

**Deletion.** To delete a record, the system first looks up the record to be deleted. The actions the system takes next **depend on whether the index is dense or sparse:**

- **Dense indices:**

1. If the **deleted record was the only record** with its particular search-key value, then the system deletes the corresponding index entry from the index.
2. **Otherwise** the following actions are taken:
  - a. If the index **entry stores pointers to all records (Dense non-clustering)** with the same search key value, the system deletes the pointer to the deleted record from the index entry.
  - b. **Otherwise, the index entry stores a pointer to only the first record (Dense Clustering)** with the search-key value (i.e. Dense cluster index). In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.





**1.If Delete <98345, Kim, Ele. Eng. 80000>** , as there is only one entry corresponding to **Ele. Eng** in index , delete the record from the file and also delete Ele. Eng. Index entry.

**2b. If Delete <10101, Srinivasan, 'Comp. Sci., 65000>** and this is the record to which Comp.Sc index entry is pointing, delete the record from the file and pointer is updated to point **<45565, Katz, Comp. Sci., 75000>**

Index table  
based on Salary

40000	
60000	
62000	
65000	
72000	
75000	
80000	
87000	
90000	
92000	
95000	



ID      Name      Discipline      Salary

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



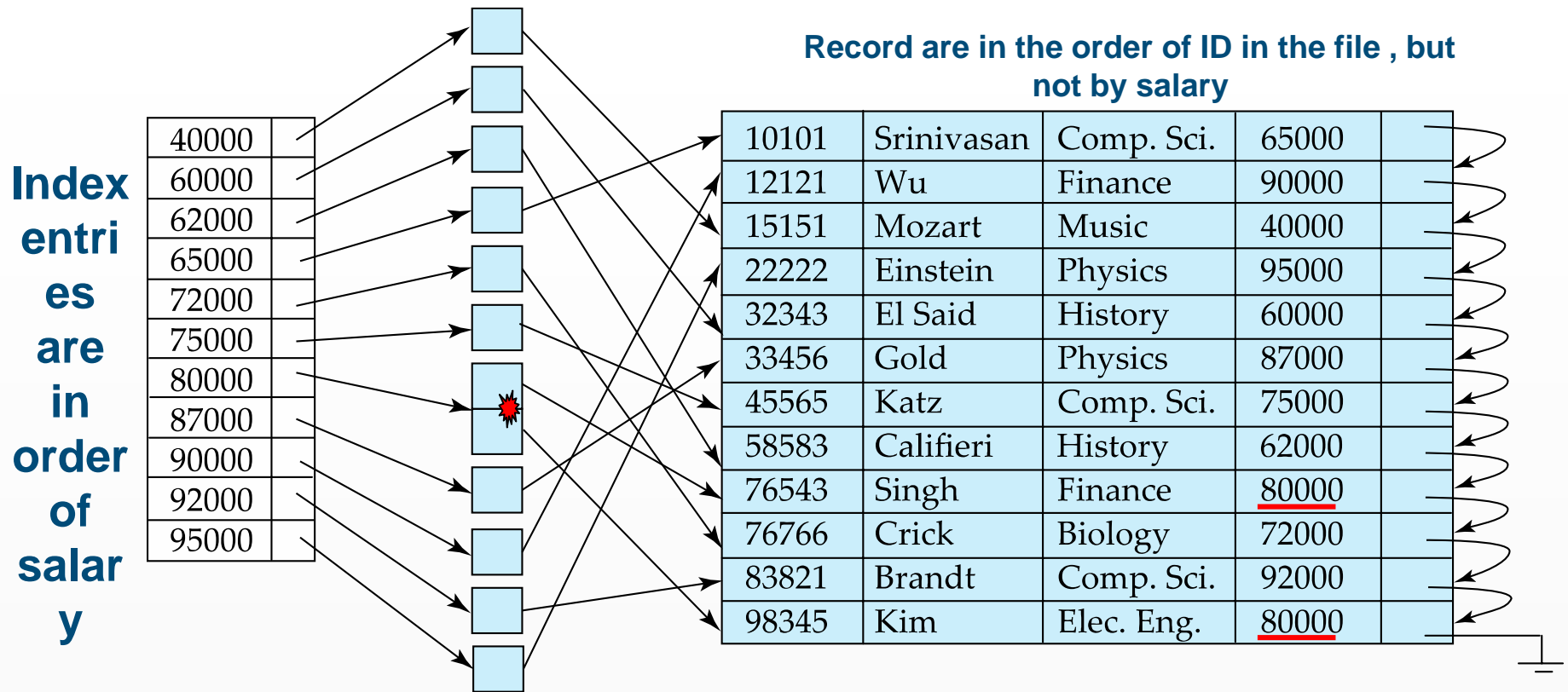
**2a.** If Delete **<15151,Mouzart,Music,40000>** then delete the record from file and also index entry corresponding to 15151

**2a.search\_key-** Salary points to every record in the file , means if 80000 in index points to all records i.e. 76543 and 98345 then assume we are deleting **<76543,...> , delete record from file and also pointer from index , so index contains only pointers to <98345,... >**

# 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.
- Assume that, in the *instructor* relation, records are stored sequentially by ID, also assume that there may be an index created on search\_key ID. i.e. Primary index on ID.
  - **Example 1:** We may want to find all instructors in a particular department.
  - **Example 2:** We want to find all instructors with a specified salary or with salary in a specified range of values.
- In the above two examples, we may need two different index files on Dept\_name , Salary respectively. In both index files, Dept\_name values are sorted order(1<sup>st</sup> index) and Salary values in sorted order in (2<sup>nd</sup> Index file). However actual Instructor record are in different order(by ID) than Dept\_name or Salary.
- We can have two separate secondary index for each search-key value (dept\_name as well as Salary)

# 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**

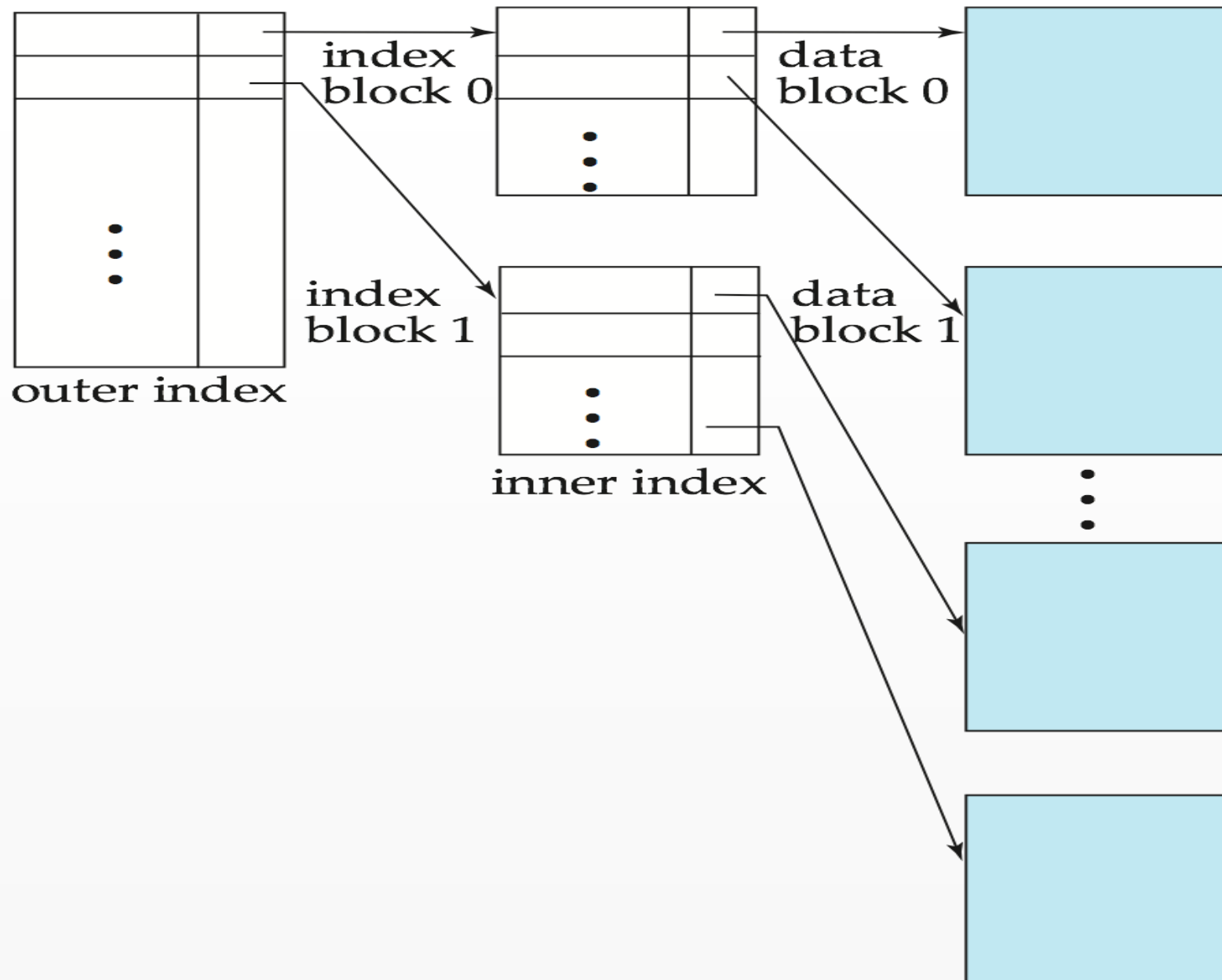
# Primary and Secondary Indices

- 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**.
  - Because secondary-key order and physical-key order differ, if we attempt to **scan** the file **sequentially in secondary-key order**, the reading of each record is likely to require the reading of a new block from disk, which is very slow. 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.)





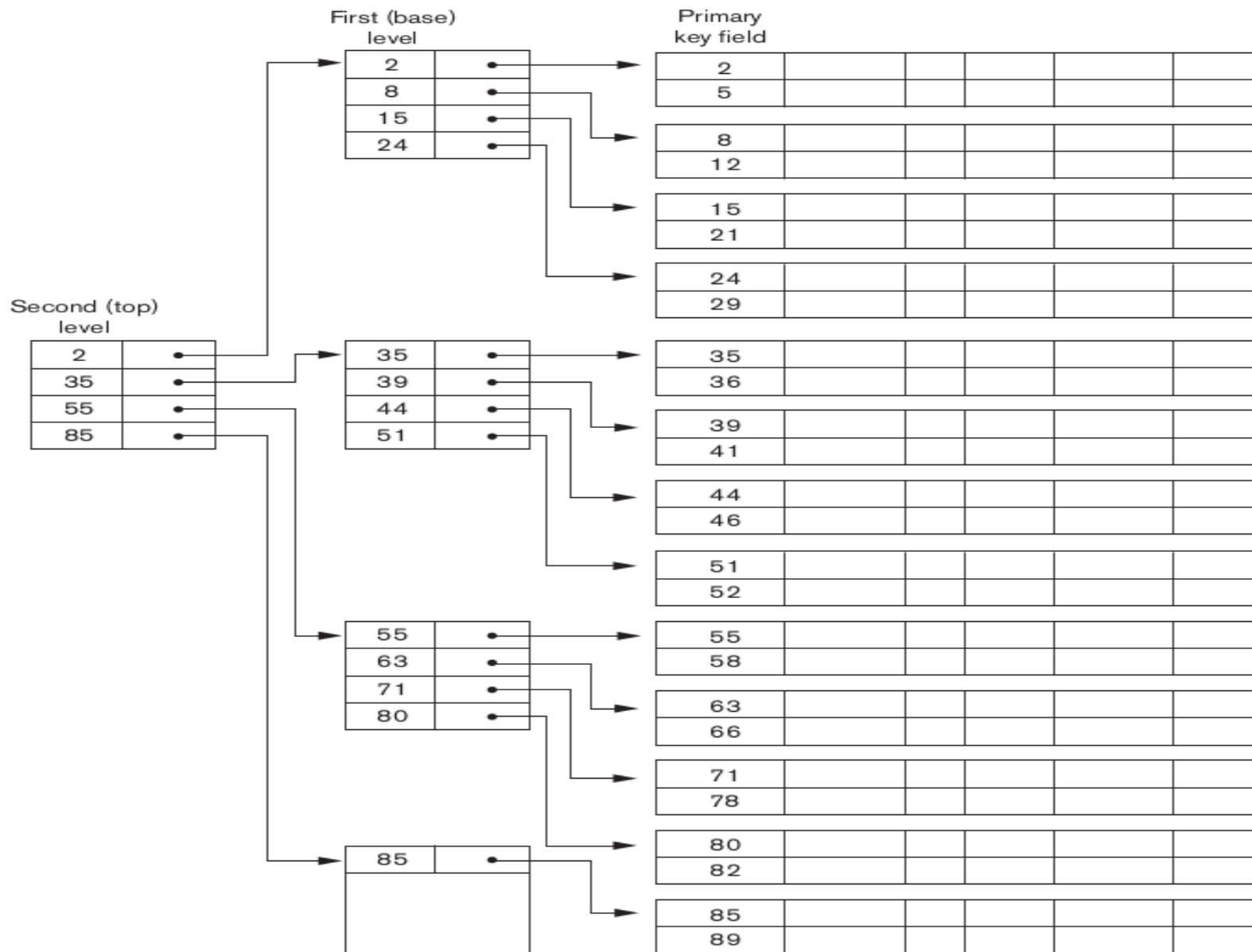
# Example: Multilevel Index

**Figure 18.6**

A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.

**Two-level index**

**Data file**



# 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 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

- There are 10 buckets,
- The term bucket to denote a unit of storage that can store one or more records. A **bucket is typically a disk block**.
- but could be chosen to be **smaller or larger than a disk block**.
- Assume  $h(K_i)$  is the hash function and  $K_i$  is the search key value.
- Assume following are the has values obtained when  $h(\text{dept\_name})$  is done.
  - E.g.  $h(\text{Music}) = 1$   $h(\text{History}) = 2$   
 $h(\text{Physics}) = 3$   $h(\text{Elec. Eng.}) = 3$

# 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


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

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

# Hash Functions

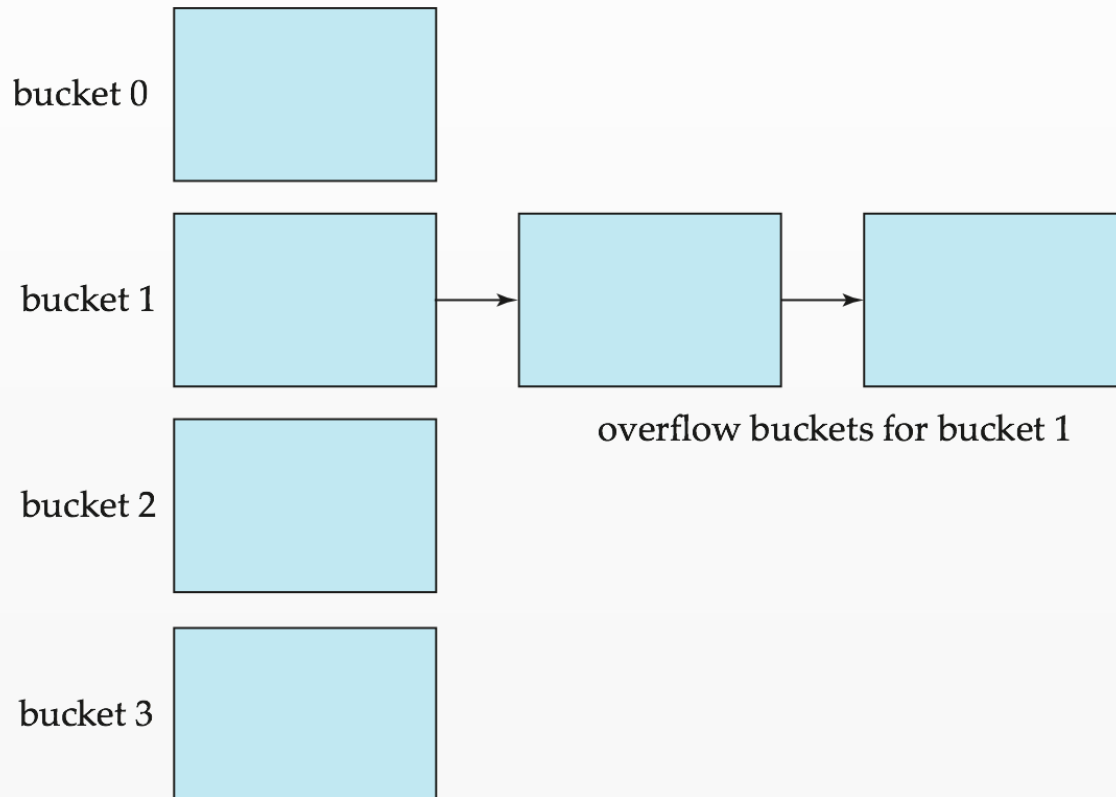
- ❑ **Worst hash function** maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- ❑ An **ideal hash function** is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- ❑ Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- ❑ Typical hash functions **perform computation on the internal binary representation of the search-key**.
  - ❑ For example, for a string search-key, the **binary representations of all the characters in the string could be added and the sum modulo the number of buckets** could be returned. .

# Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - **Skew in distribution** of records. This can occur due to two reasons:
    - ▶ multiple records have same search-key value
    - ▶ **chosen hash function** produces non-uniform distribution of key values
- Although the probability of **bucket overflow** can be **reduced**, it **cannot be eliminated**;
- It is handled by using *overflow buckets*.

# Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.



# Handling of Bucket Overflows

- ❑ Bucket overflow can occur because of
  - ❑ Insufficient buckets
  - ❑ **Skew in distribution** of records. This can occur due to two reasons:
    - ▶ multiple records have same search-key value
    - ▶ **chosen hash function** produces non-uniform distribution of key values
- ❑ Although the probability of **bucket overflow** can be **reduced**, it **cannot be eliminated**;
- ❑ It is handled by using *overflow buckets*.



# Handling of Bucket Overflows (Cont.)

- ❑ **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- ❑ Above scheme is called **closed hashing**.

