File Organization, Record Organization

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

Fixed-Length Records

Assume that each record of Instructor file is defined (in pseudocode) as

```
type instructor = record
        ID varchar (5);
        name varchar(20);
        dept_name varchar (20);
        salary numeric (8,2);
    end
```

The *instructor* record is **53 bytes** long

How to allocate blocks to store records ?

Simple approach

If the **block size** is **not** happens to be a **multiple of 53**, then some records will cross block boundaries.

It is **difficult to delete** a record from this structure.

To avoid the first problem, we allocate **only as many records to a block as would fit entirely in the block** (this number can be computed easily by **dividing the block size by the record size**, and discarding the fractional part). Any remaining bytes of each block are left unused.
If **block size** is 120 Bytes , then store only **2 records (2*53=106**) remaining **14** bytes **unused.**

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. (assuming ***i*** starts with 0, **n*i** gives starting byte offset address)
    - Ex: *i=0* , offset bytes for **0th record is 0** , *i=1* offset bytes for 1st record is 1*53=53 byte, starting offset byte for 2nd record is 2*53=106th byte.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries

- **Deletion of record** *i:*
  alternatives*:*
  1. move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  2. move record n^{th} to i^{th}
  3. 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

1. 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

move records $i + 1, \ldots, n$ **to** $i, \ldots, n - 1$

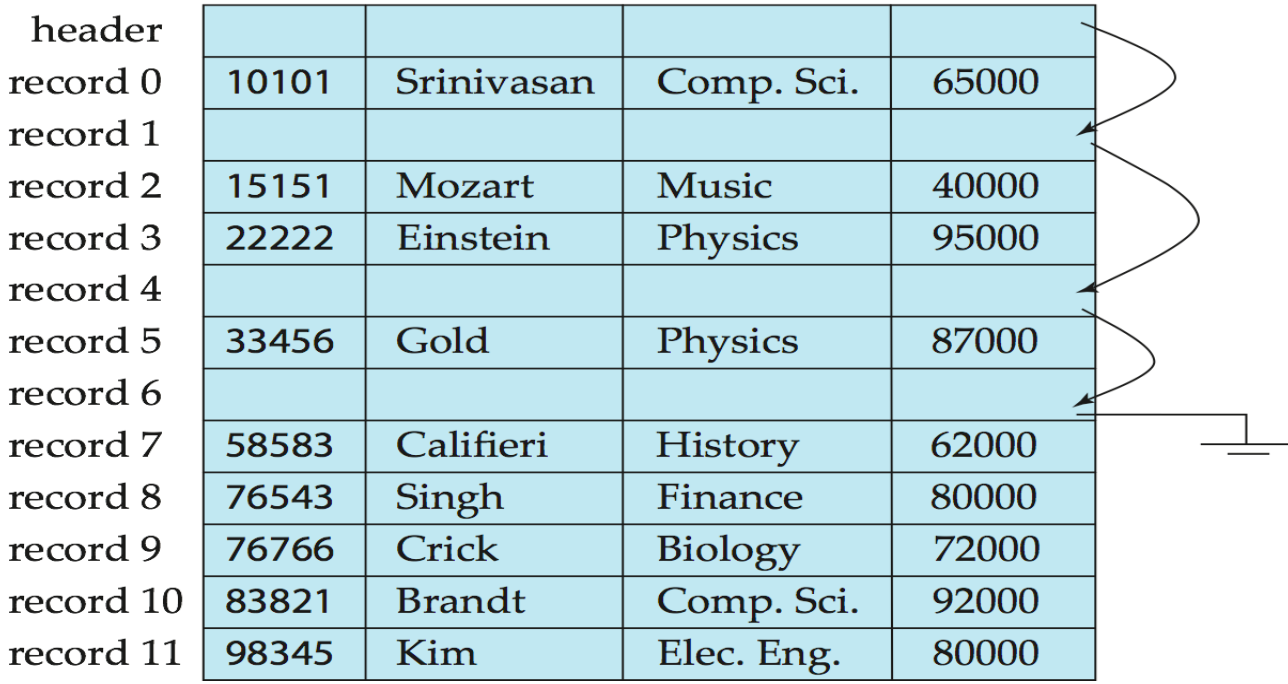
2. Deleting record 3 and moving last record

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

move record *n* **to** *i*

3. 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
- The deleted records thus form a **linked list**, which is often referred to as a **free list**
- On insertion of a new record, we use the record pointed to by the header and change the header pointer to point to the next available record. of the file.
- If no space (free list empty) then add the new record to the end.



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

- **Why Variable-length records ?**
  - 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)**
- **Problems needs to be addressed:**
  - How to represent attributes in 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

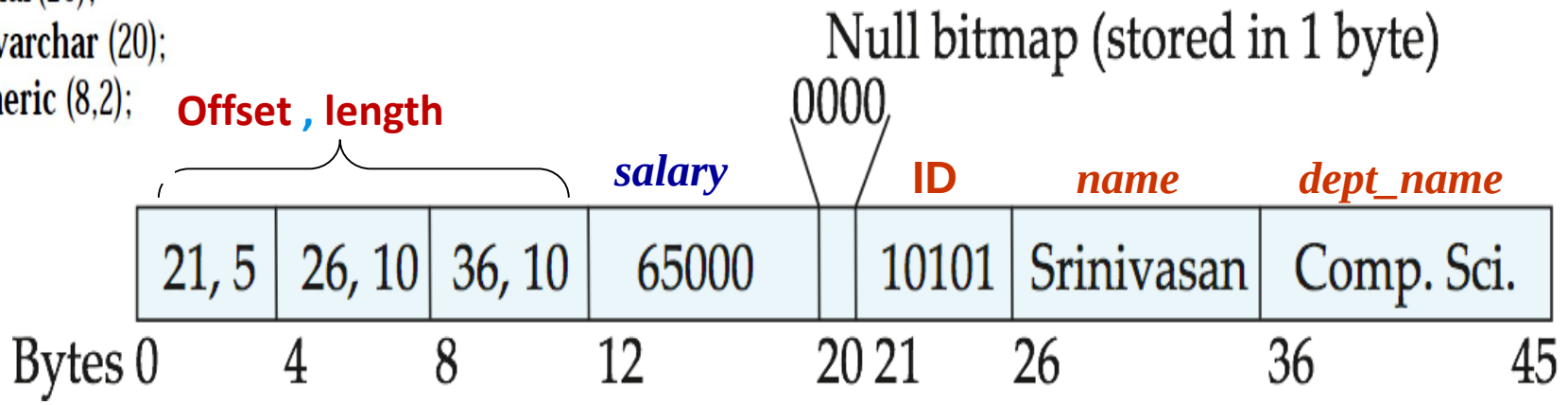
Representation of records with variable length attributes

Addressing-How individual attributes are extracted

- 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.**
- In the initial part of record, 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

```
type instructor = record
        ID varchar (5);
        name varchar(20);
        dept_name varchar (20);
        salary numeric (8,2);
    end
```



The figure shows an ***instructor*** record, whose first three attributes ***ID, name***, and ***dept_name*** are **variable-length** **strings**, and whose fourth attribute ***salary*** is a **fixed-sized** number.

A **null bitmap**, which indicates which attributes of the record have a null value. Above example **4 bits** are used because there are **4 attributes**.

In this particular record, if the **salary were null**, the **1st bit** of the bitmap would be **set to 1** i.e. 1000(while reading 12th to 19th bytes ignored)

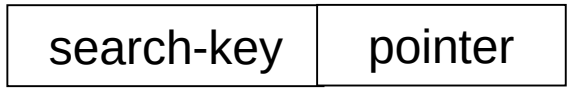
Indexing

Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

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



- 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*

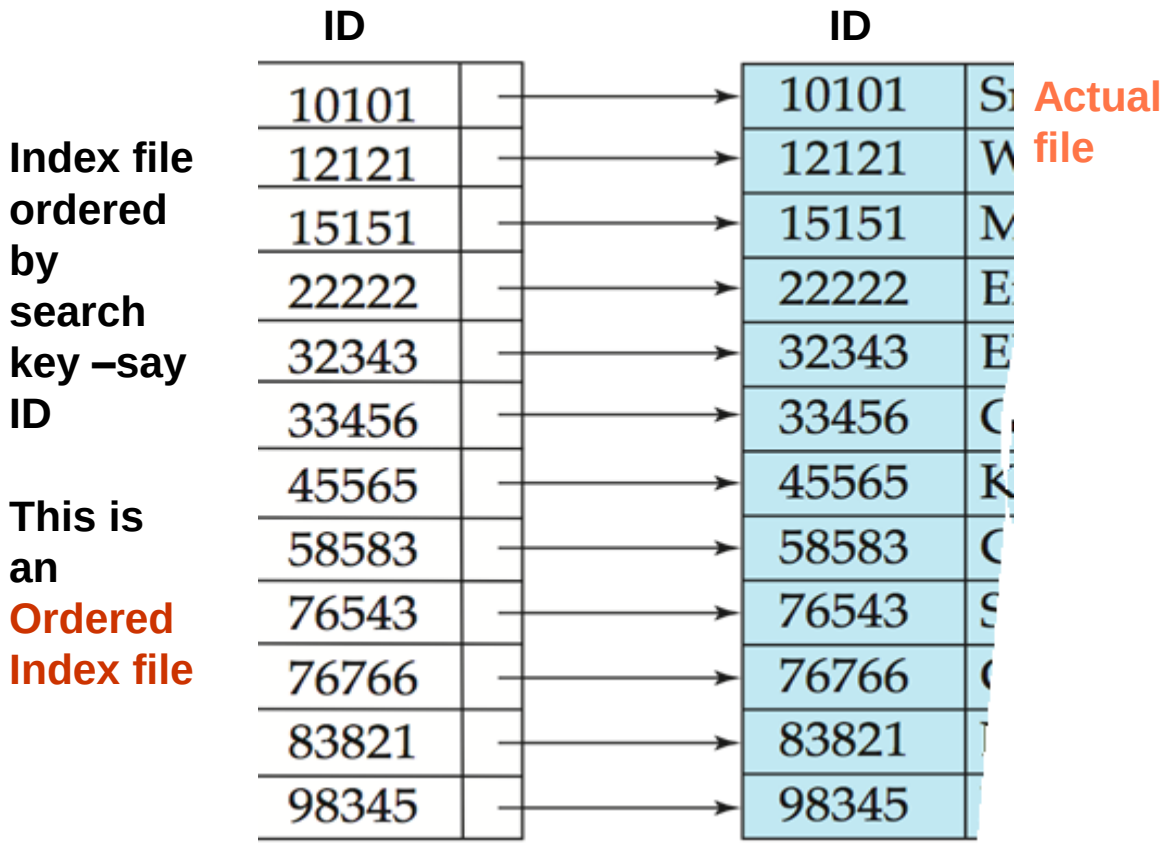
- **Access time-** time to find a particular data item, or set of items
- **Insertion time-** time to find the correct place to insert the new data, to update the index structure
- **Deletion time -** time to takes to find the item to be deleted , to update the index structure.
- **Space overhead-**additional 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. slide
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file. slide
  - 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.** Discussed latter in slide
- **Index-sequential file:** ordered sequential file with a primary index.

Ordered Indices

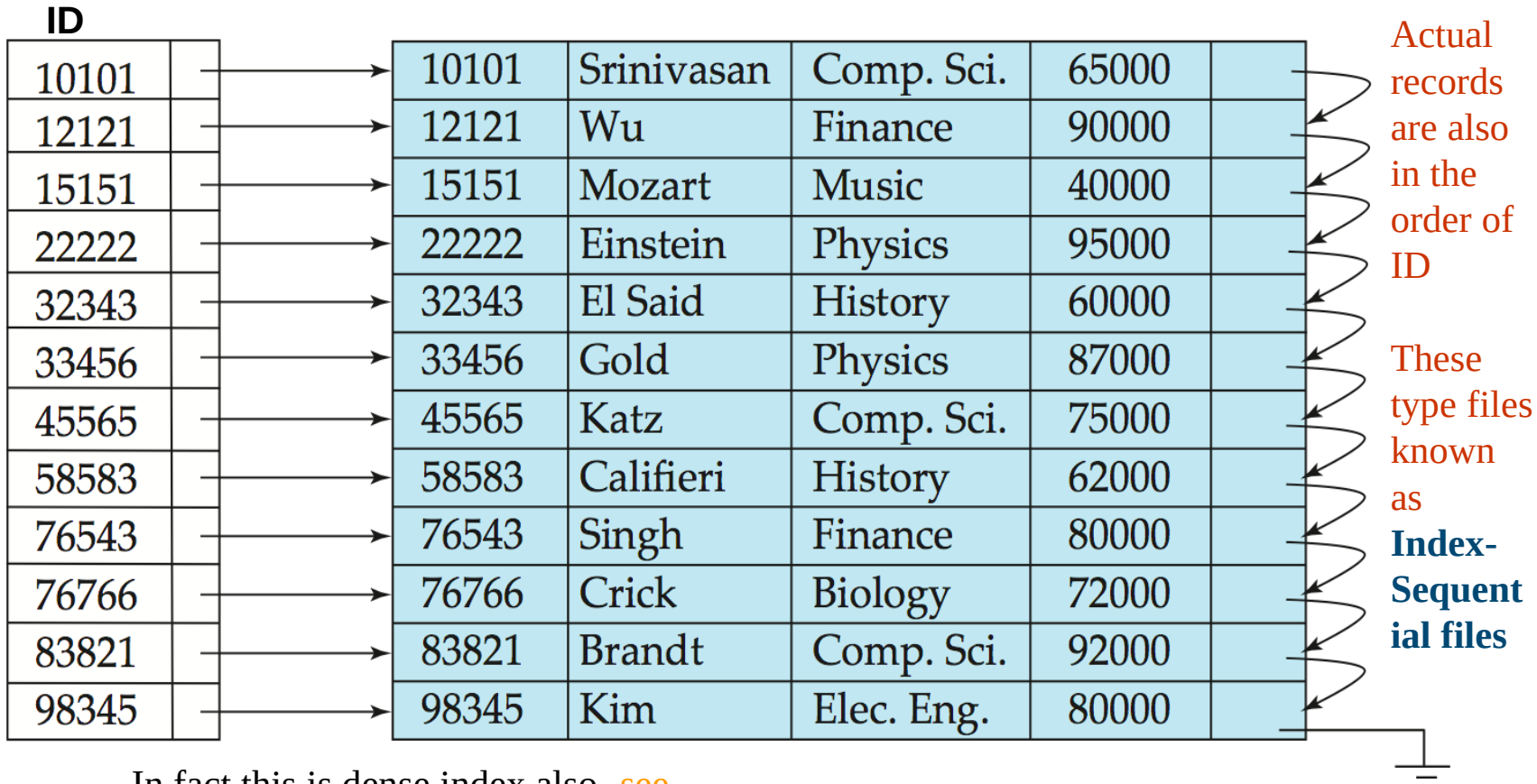
- In an **ordered index,** index entries in the **index file** are stored sorted on the **search key value**.



Ordered Indices (Contd..)

- **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.

Index is created on ID attribute of Instructors which is in the sorted order



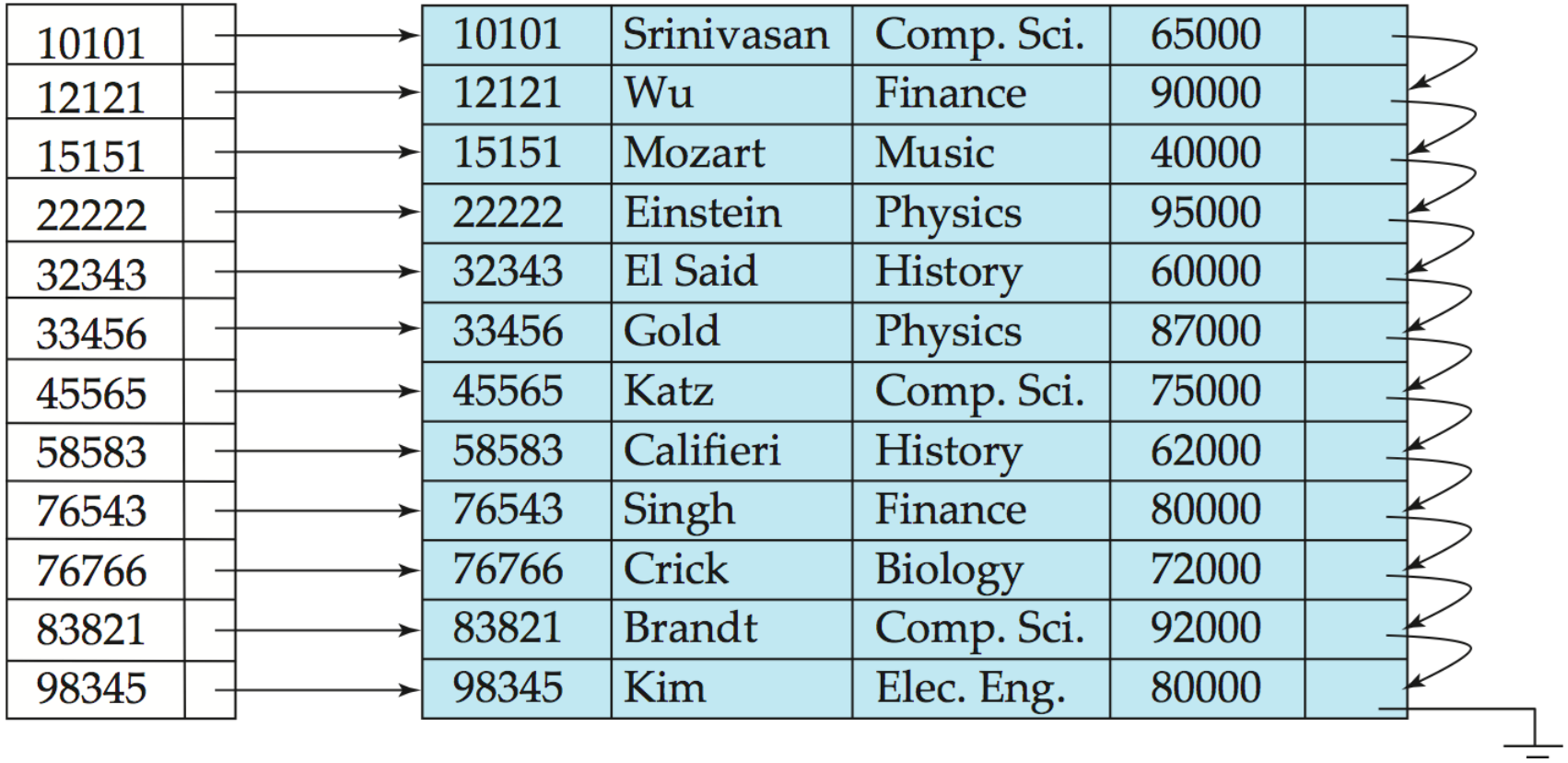
Actual records are also in the order of ID

These type files known as **Index-Sequential files**

In fact this is dense index also- see

Dense Index Files

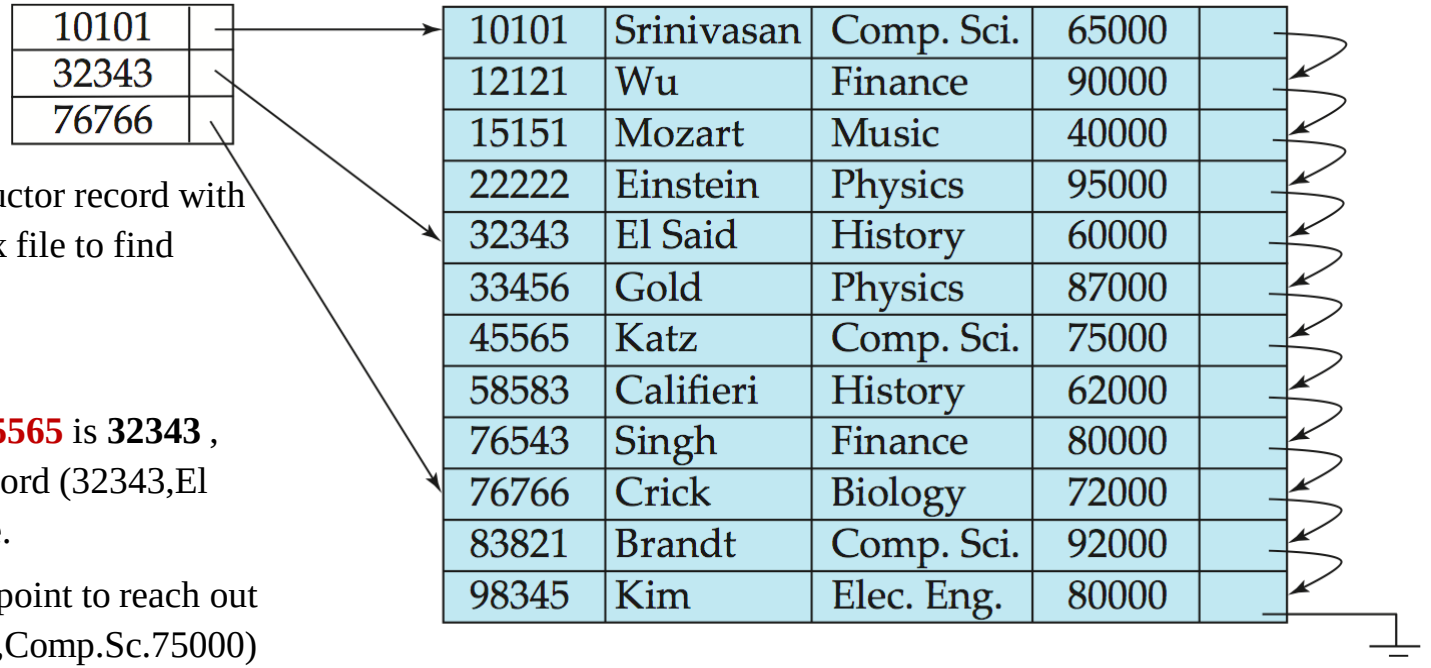
- **Dense index** — Index record appears for every kind of search-key value (for every ID value) in the file. Search Key value may be Unique/Duplicate
- E.g. index on *ID(**unique**)* attribute of *instructor* relation



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

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



Assume we want search Instructor record with ID K=45565. Search the index file to find largest search key value < K.

i.e. 10101< 32343 < 45565

largest search key value < 45565 is **32343** , follow the pointer to reach record (32343,El Said,History,60000) in the file.

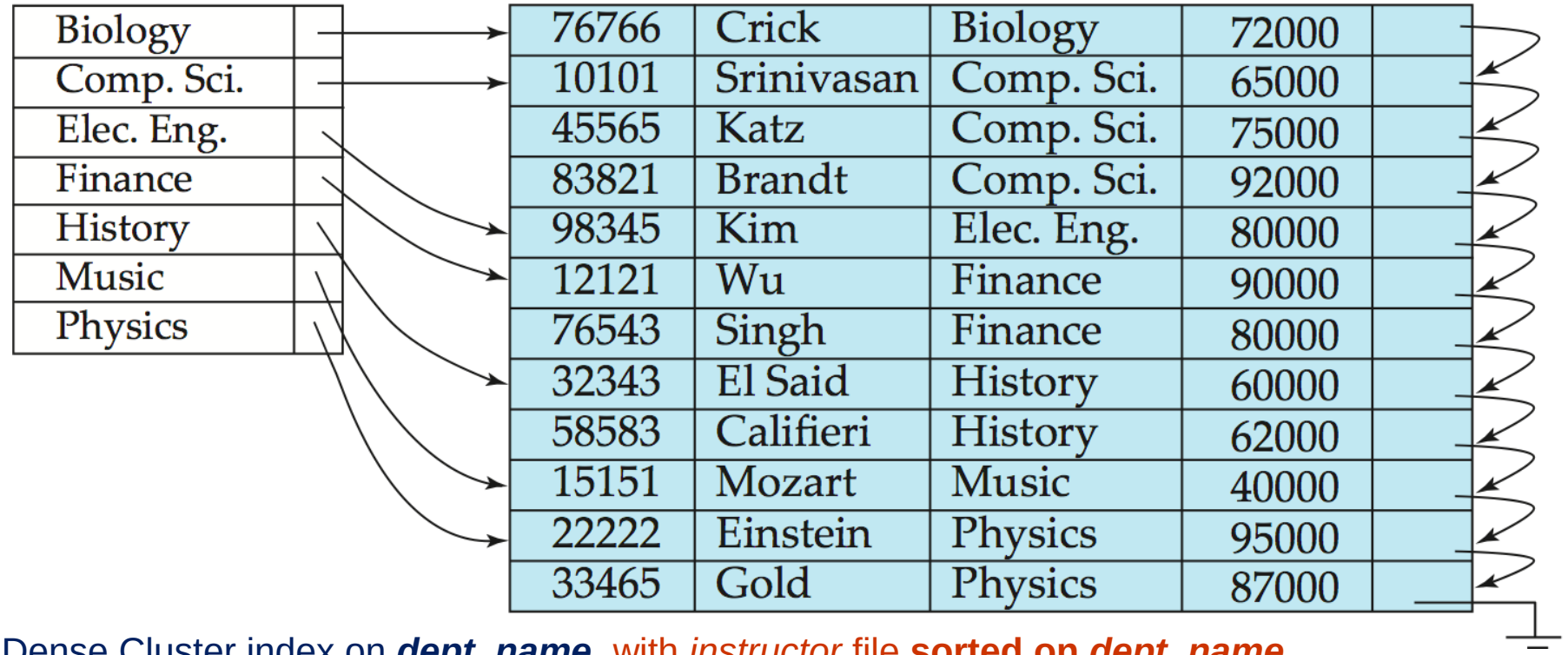
Search sequentially from this point to reach out required account (45565,Katz,Comp.Sc.75000)

Dense Cluster Index Files

- In a **dense clustering** index, the index record contains the search-key value entry for every kind of search key 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.(one index entry for a cluster(group) of records with same search key value-one pointer to cluster)

Dense- Every type of search key value(Dept_Name values) has an entry in index file.

Cluster- Pointer is to group of record having same search value



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

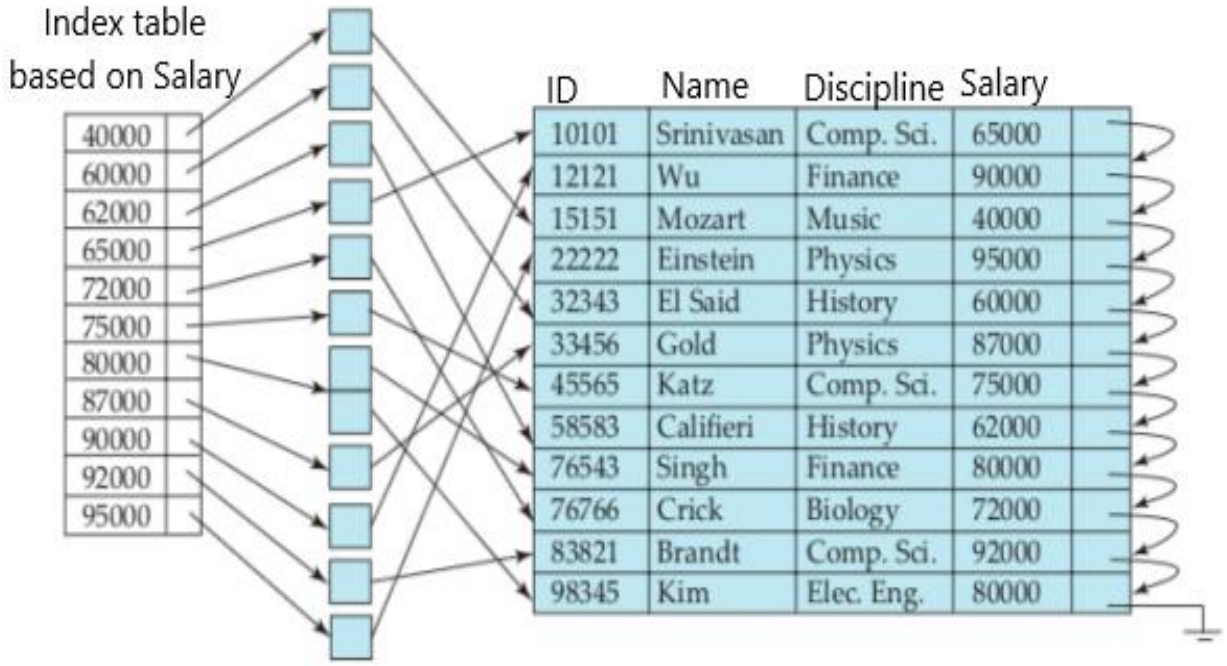
In a **dense non-clustering index**, the index must store a **list of pointers to all records** with the same search-key value.

Dense Non-Cluster Index Files

In a **dense non-clustering index**, the index must store a **list of pointers to all records** with the same search-key value. (Multiple records with same key value exist-search key not unique but one index entry per kind of search key value but pointer to all records with same key)

Dense- Every type of search key value(Dept_Name values) has an entry in index file.

Non-Cluster- Pointer is to every record having same search value

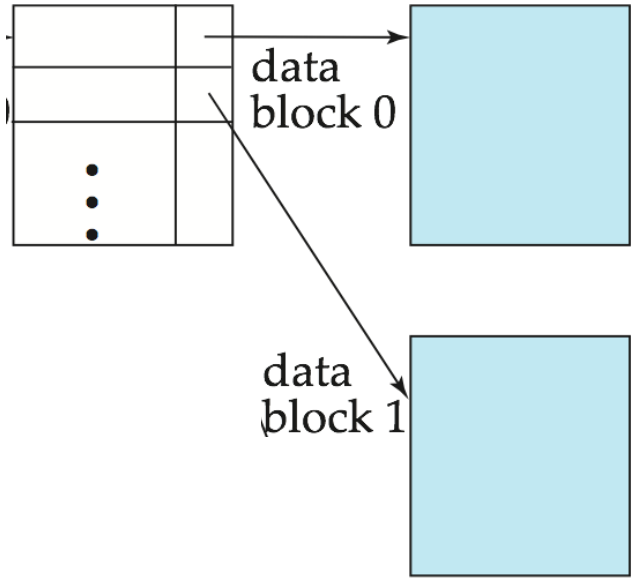


In the above example , we can see an index entry Comp.Sc is having list of 3 pointers pointing to Instructors with IDs 10101,45565,83821. Similarly Index entry Finance is having list of 2 pointers pointing to Instructor records with IDs-12121,76543, Similarly we can see list of pointers for index entry History & Physics also.

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.**

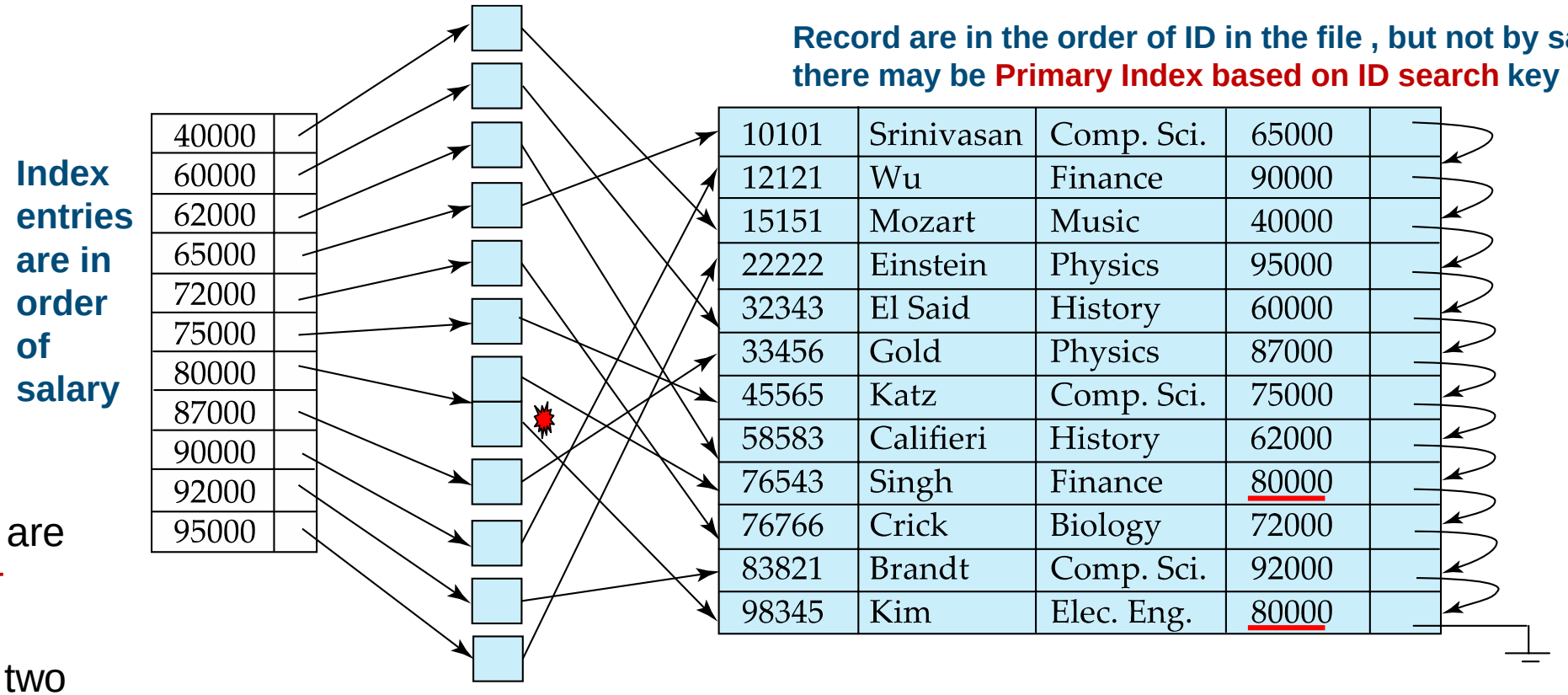
Bringing one Block at a time from Hard disk (involves 1 seek & 1 Rotational Delay) to memory and searching required key in the block (Searching in memory less time)



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^{st} index) and Salary values in sorted order in (2^{nd} 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*

- Note there are two pointer entries pointing to two records having same salary value 8000

- 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**

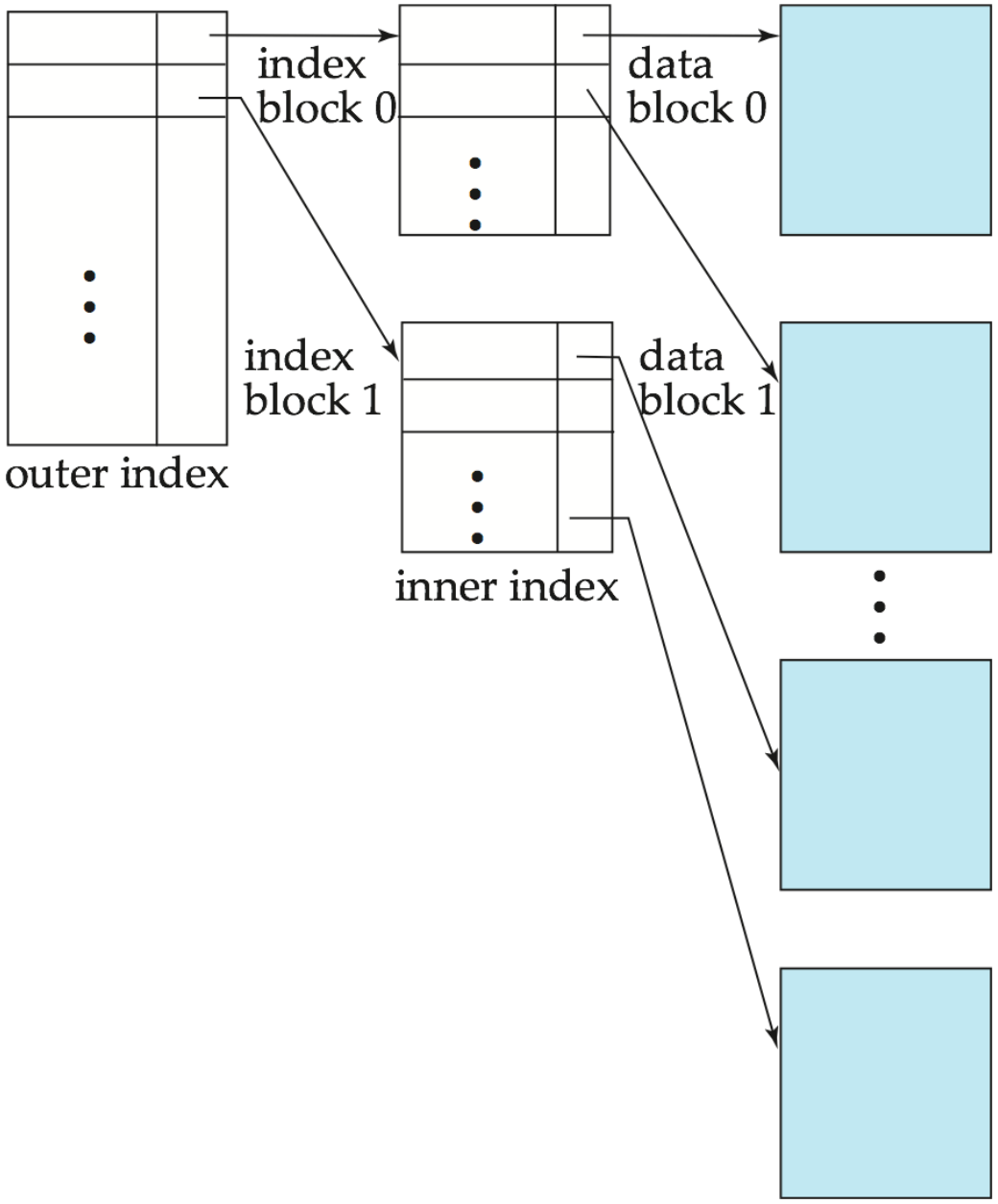
Primary and Secondary Indices

- 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

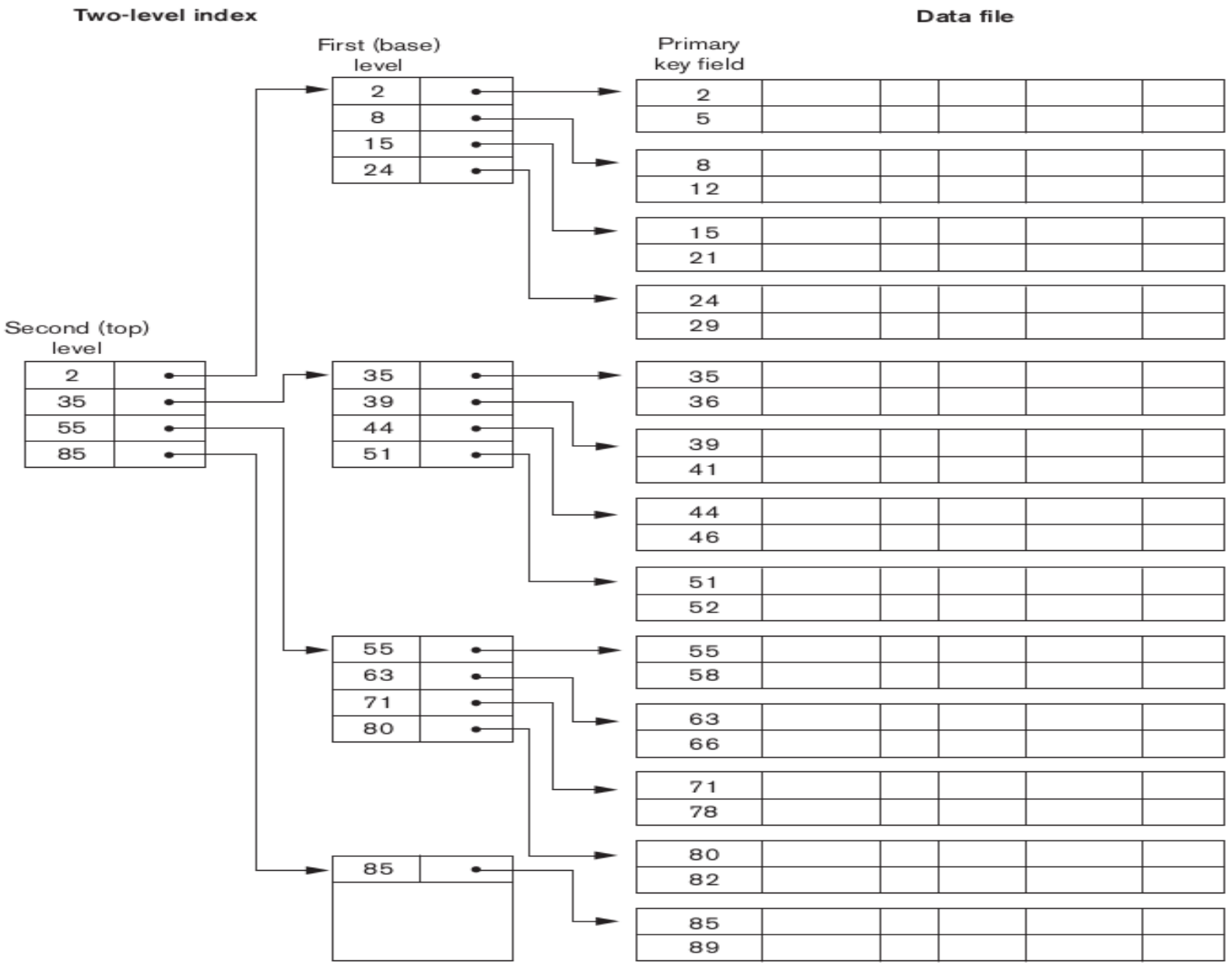
- If primary index does not fit in memory, access becomes expensive.
- **Solution:** treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large 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.



Transactions

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes while executing transactions.
  - Concurrent execution of multiple transactions without inconsistency.

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items.

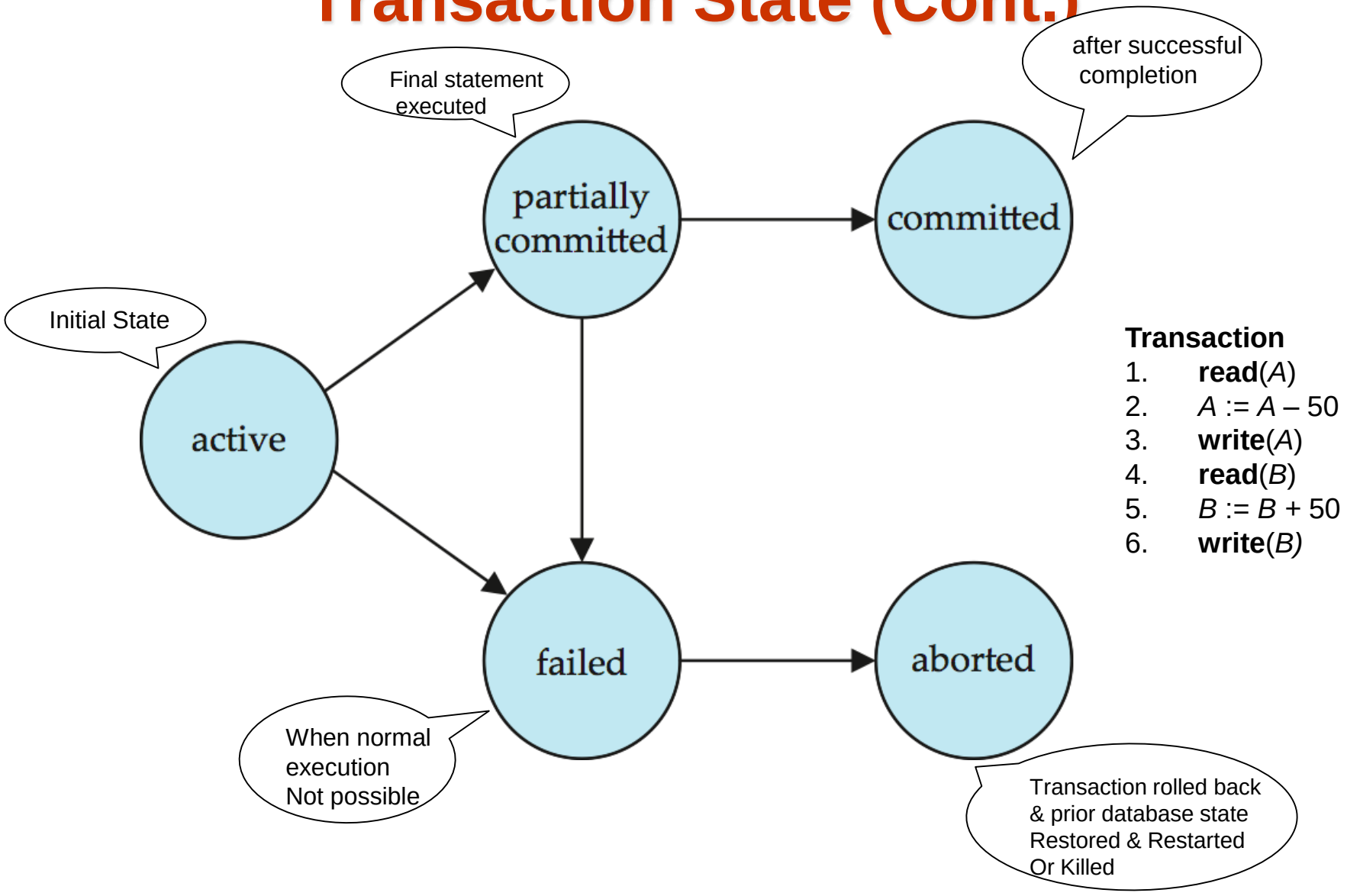
To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all transaction operations are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed –** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Transaction

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Schedules

- **Schedule** – A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed.
  - a schedule for a set of transactions must consist of all instructions of those transactions.
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

Schedule 1 (Serial)

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.
- Suppose the current values of accounts *A* and *B* are \$1000 and \$2000.
- A **serial schedule** in which T_1 is followed by T_2 :

The final values of accounts A and B, after the execution in Schedule 1 takes place, are \$855 and \$2145, respectively.

Thus, the total amount of money in accounts A and B—that is, the **sum A + B —is preserved** after the execution of both transactions.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

At the end of T1 A=950 and B=2050
At the end of T2 A=855 and B=2145

Total Amount in the system is 2145+855=3000
Database consistency maintained.

Schedule 2 (Serial)

- A serial schedule where T_2 is followed by T_1

Assume current value of A=1000 and B=2000

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is in schedule 2

Again, as expected, the **sum A + B is preserved**, and the final values of accounts A and B are $850 and $2150, respectively.

At the end of T2 A=900 and B=2100
At the end of T1 A=850 and B=2150

Total Amount in the system is 2150+850=3000
Database consistency maintained.

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Schedule 3 (Concurrent)

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

accounts *A* and *B* are $1000 and $2000.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Finally A=855 and B=2145
; A+B=855+2145=3000
–consistency maintained

In Schedules 1, 2 and 3, the **sum A + B is preserved**.

Schedule 4 (Concurrent)

- The following concurrent schedule does not preserve the value of $(A + B)$. (Assume initially A=1000/- and B=2000/-)

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

A=1000
1000-50=950
A =950
B=2000
B=2000+50=2050
B=2050

A =1000
Temp=100
1000-100=900
A=900
B=2000
2000 +100=2100
B=2100

How to check which kind of concurrent Schedule maintain Consistency

Final A=950 and B=2100, now A+B =3050 which violates Consistency

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly **concurrent**) **schedule** is serializable if it is **equivalent to** a **serial schedule**.
- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**