Module iv

DATA STORAGE AND QUERYING

RAID(redundant arrays of independent disks

- The data-storage requirements of some applications (in particular Web, database,and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk-drive capacities have been growing very fast.
- Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Several independent reads or writes can also be performed in parallel

- A variety of disk-organization techniques, collectively called redundant arrays of independent disks (RAID), have been proposed to achieve improved performance and reliability.
- RAID (redundant array of independent disks) is a way of storing the same data in different places on multiple hard disks or solid-state drives to protect data in the case of a drive failure.

- Store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information
- Effective mean time to failure is increased
- Simplest (but expensive) approach to redundancy is to duplicate every disk.
- This technique is called ***mirroring*** (**shadowing**).

- A logical disk then consists of two physical disks, and every write is carried out on both disks.
- If one of the disks fails, the data can be read from the other.Data will be lost only if the second disk fails before the first failed disk is repaired.
- With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

- With multiple disks, we can improve the transfer rate as well (or instead) by **striping** data across multiple disks. In its simplest form, **data striping** consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**.

Different type of data striping :

1. **Bit level striping** : Splitting the bits of each byte across multiple disks

   : No of disks either is a multiple of 8 or a factor of 8

   : These disks are considered as single disk.

   E.g. : Array of eight disks, write bit i of each byte to disk I

2. **Block-level striping** **:** Stripes blocks across multiple disks

   : Fetches n blocks in parallel from the n disks

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- **Block-level striping** stripes blocks across multiple disks. It treats the array of disks as a single large disk, and it gives blocks logical numbers; we assume the block numbers start from 0. With an array of n disks, block-level striping assigns logical block i of the disk array to disk (i mod n) + 1; it uses the (i /n)th physical block of the disk to store logical block i . For example, logical block 11 is stored in physical block 1 of disk 4.

- When reading a large file, block-level striping fetches n blocks at a time in parallel from the n disks, giving a high data-transfer rate for large reads.
  When a single block is read, the data-transfer rate is the same as on one disk, but the remaining n − 1 disks are free to perform other actions.

- In summary, there are two main goals of parallelism in a disk system:
  1. Load-balance multiple small accesses (block accesses), so that the throughput of such accesses increases.
  2. Parallelize large accesses so that the response time of large accesses is reduced.

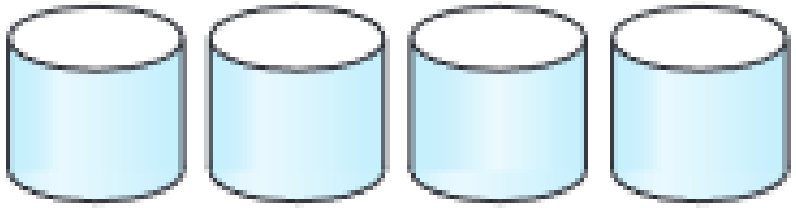
RAID Levels

- Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but does not improve reliability.
- Various alternative schemes aim to provide redundancy at lower cost by combining disk striping with “parity” bits
- These schemes have different cost—performance trade-offs. The schemes are classified into RAID levels
- (For all levels, the figure depicts four disks’ worth of data, and the extra disks depicted are used to store redundant information for failure recovery.)

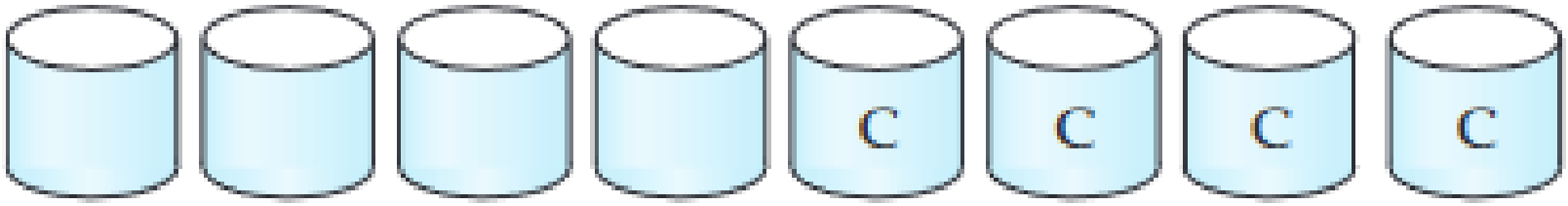
- **RAID level 0** refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits).
- **RAID level 1** refers to disk mirroring with block striping.
- **RAID level 2**, known as memory-style error-correcting-code (ECC) organization, employs parity bits. Memory systems have long used parity bits for error detection and correction

- Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte that are set to 1 is even (parity = 0) or odd (parity = 1).
- If one of the bits in the byte gets damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity.
- Similarly,if the stored parity bit gets damaged, it will not match the computed parity.Thus, all 1-bit errors will be detected by the memory system. Error-correcting schemes store 2 or more extra bits, and can reconstruct the data if a single bitgets damaged.

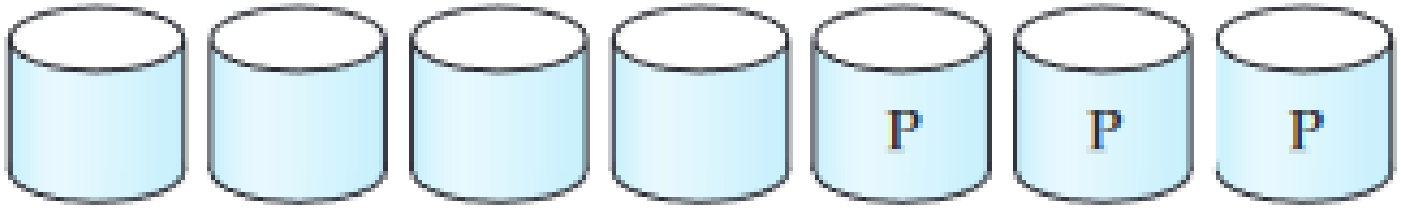
- The idea of error-correcting codes can be used directly in disk arrays by striping bytes across disks. For example, the first bit of each byte could be
stored in disk 0, the second bit in disk 1, and so on until the eighth bit is stored in disk 7, and the error-correction bits are stored in further disks.



(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



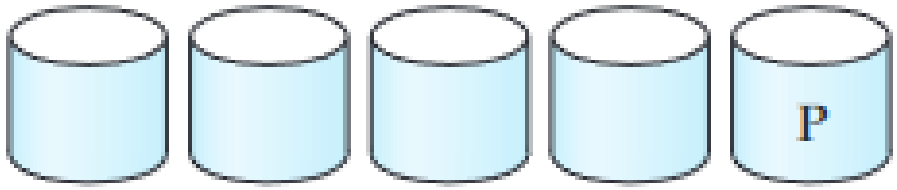
(c) RAID 2: memory-style error-correcting codes

- The disks labeled P store the error correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks, and can be used to reconstruct the damaged data.

- **RAID level 3**, bit-interleaved parity organization, improves on level 2 by exploiting the fact that disk controllers, unlike memory systems, can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection.
- If one of the sectors gets damaged, the system knows exactly which sector it is, and, for each bit in the sector, the system can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1

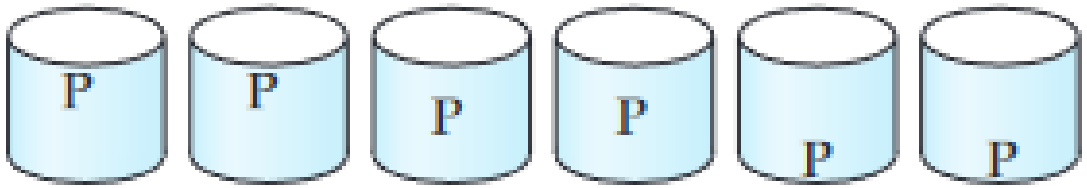
- RAID level 3 is as good as level 2, but is less expensive in the number of extra disks (it has only a one-disk overhead), so level 2 is not used in practice.
- RAID level 3 has two benefits over level 1. It needs only one parity disk for several regular disks, whereas level 1 needs one mirror disk for every
disk, and thus level 3 reduces the storage overhead.

(d) RAID 3: bit-interleaved parity



(e) RAID 4: block-interleaved parity

(f) RAID 5: block-interleaved distributed parity



(g) RAID 6: P + Q redundancy

- **RAID level 4**, block-interleaved parity organization, uses block-level striping, like RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from N other disks.
- If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk

- **RAID level 5**, block-interleaved distributed parity, improves on level 4 by partitioning data and parity among all N + 1 disks, instead of storing data in
  N disks and parity in one disk.

- **RAID level 6**, the P + Q redundancy scheme, is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures.
  Instead of using parity, level 6 uses error-correcting codes.

File Organization

- A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks.
- A file is organized logically as a sequence of records. These records are mapped onto disk blocks.
- Each file is also 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

- A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used.
- In a relational database, tuples of distinct relations are generally of different sizes. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records;

Fixed-Length Records

- As an example, let us consider a file of instructor records for our university database. Each record of this file is defined (in pseudocode) as:

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

- Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes.
- Suppose that instead of allocating a variable amount of bytes for the attributes ID, name, and dept name, we allocate the maximum number of bytes that each attribute can hold.
- Then, the instructor record is 53 bytes long. A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on

- However, there are two problems with this simple approach:
- 1. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.
  2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored

- To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block.
- When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead .
- Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record

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

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

e 10.5 File of Figure 10.4, with record 3 deleted and all records moved.

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

e 10.6 File of Figure 10.4, with record 3 deleted and final record moved.

- It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space. A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

- At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted.
- We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as pointers, since they point to the location of a record.
- 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.
- We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.
- Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record.

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

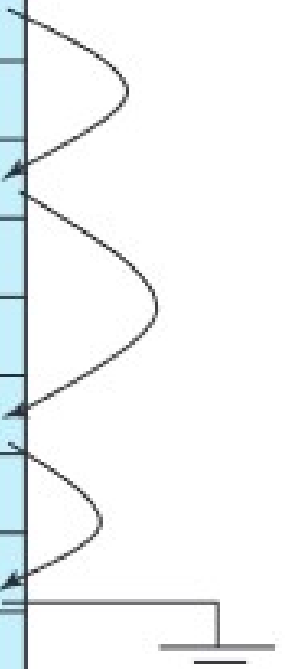


Figure 10.7 File of Figure 10.4, with free list after deletion of records 1, 4, and 6.

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.
  - Record types that allow repeating fields, such as arrays or multisets.

- Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:
  • 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

- The representation of a record with variable-length attributes typically has two parts: **an initial part** with fixed length attributes, followed **by data for variable-length attributes**.
- Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value.
- Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (offset, length), where offset denotes where the data for that attribute begins within the record, and length is the length in bytes of the variable-sized attribute.

- The values for these attributes are stored consecutively, after the initial fixed-length part of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length

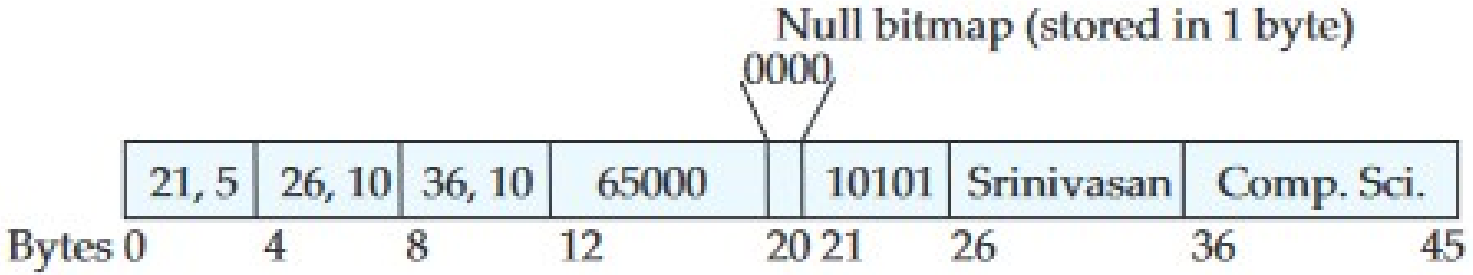


Figure 10.8 Representation of variable-length record.

- **null bitmap**, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored
  in bytes 12 through 19 would be ignored.
- The slotted-page structure is commonly used for organizing variable length records within a block

- There is a header at the beginning of each block, containing the following information:
  1. The number of record entries in the header.
  2. The end of free space in the block.
  3. An array whose entries contain the location and size of each record

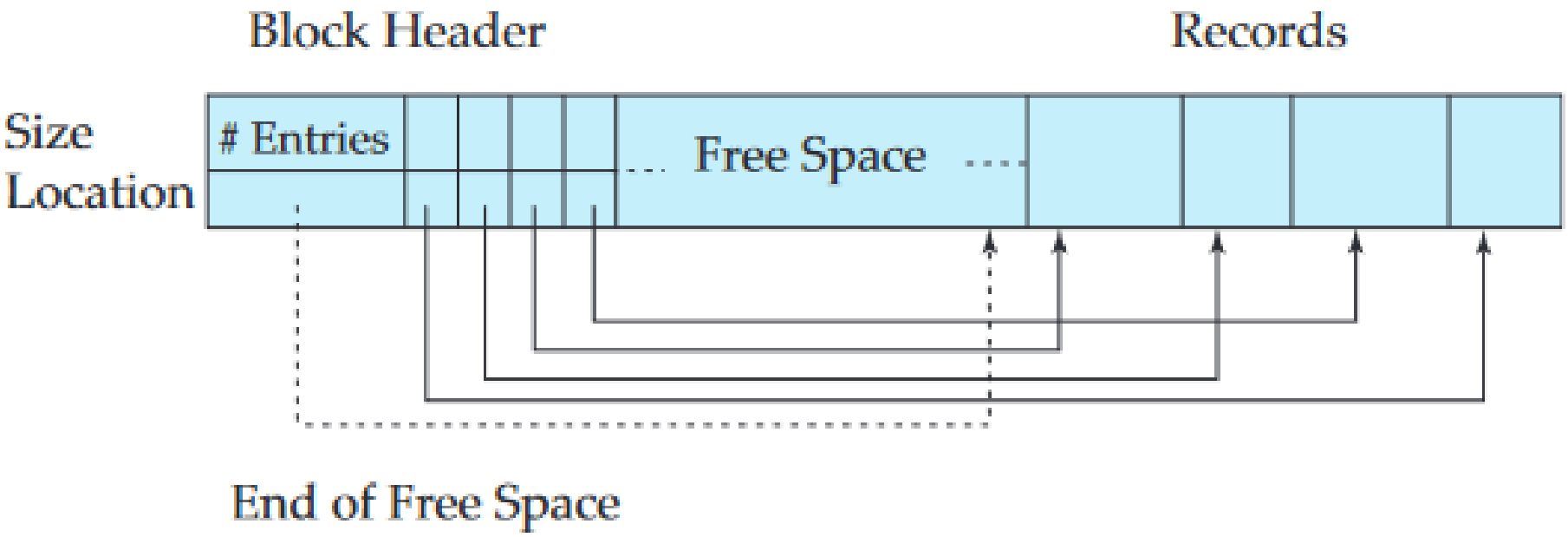


Figure 10.9 Slotted-page structure.

- The actual records are allocated contiguously in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.
- If a record is deleted, the space that it occupies is freed, and its entry is set to deleted (its size is set to −1, for example).

Organization of Records in Files

- Several of the possible ways of organizing records in files are:
- **Heap file organization.** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
- **Sequential file organization.** Records are stored in sequential order, according to the value of a “search key”of each record

- **Hashing file organization**. A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed.
- Generally, a separate file is used to store the records of each relation. However, in a multitable clustering file organization, records of several different relations are stored in the same file

Sequential File Organization

- A sequential file is designed for efficient processing of records in sorted order based on some search key.
- A search key is any attribute or set of attributes; it need not be the primary key, or even a superkey.
- To permit fast retrieval of records in search-key order, we chain together records by pointers.
- The pointer in each record points to the next record in search-key order. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

- The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms.
- For insertion, we apply the following rules:
  1. Locate the record in the file that comes before the record to be inserted in search-key order.
  2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search-key order.

multitable clustering file organization

- A multitable clustering file organization is a file organization, such as that stores related records of two or more relations in each block. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently

Indexing and Hashing

- **Basic Concepts**
- An index for a file in a database system works in much the same way as the index of textbook.
- Database-system indices play the same role as book indices in libraries. For example, to retrieve a student record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate student record.

- Keeping a sorted list of students' ID would not work well on very large databases with thousands of students, since the index would itself be very big;
  further, even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used.

- There are two basic kinds of indices:
- **Ordered indices**. Based on a sorted ordering of the values.
- **Hash indices**. Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function,
called a hash function.

- several techniques for both ordered indexing and hashing.
- Each technique must be evaluated on the basis of these factors:
- **Access types:** The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
  • **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.

- **Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.

  • **Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worth while to sacrifice the space to achieve improved performance.

- We often want to have more than one index for a file.
- An attribute or set of attributes used to look up records in a file is called a **search key**.

1.Ordered Indices

- To gain fast random access to records in a file, we can use an index structure.
- Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.
- The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute.

- file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file.
- Clustering indices are also called **primary indices**
- Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices, or secondary indices**

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	

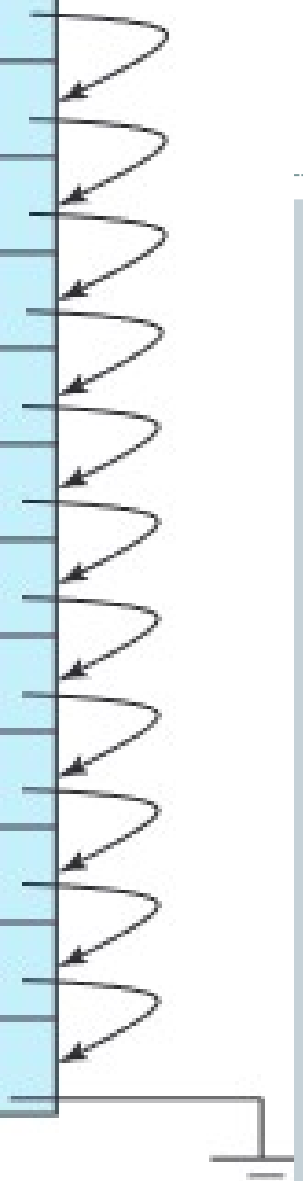


Figure 11.1 Sequential file for *instructor* records.

1.1 Dense and Sparse Indices

- An index entry, or index record, consists of a search-key value and pointers to one or more records with that value as their search-key value.
- The pointer to a record consists of the identifier of a disk block and an offset within the disk block
  to identify the record within the block.
- There are two types of ordered indices that we can use:

- **Dense index:** In a dense index, an index entry appears for every search-key value in the file. 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, since, because the index is a clustering one, records are sorted on the same search key. In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value

- **Sparse index:** In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index.

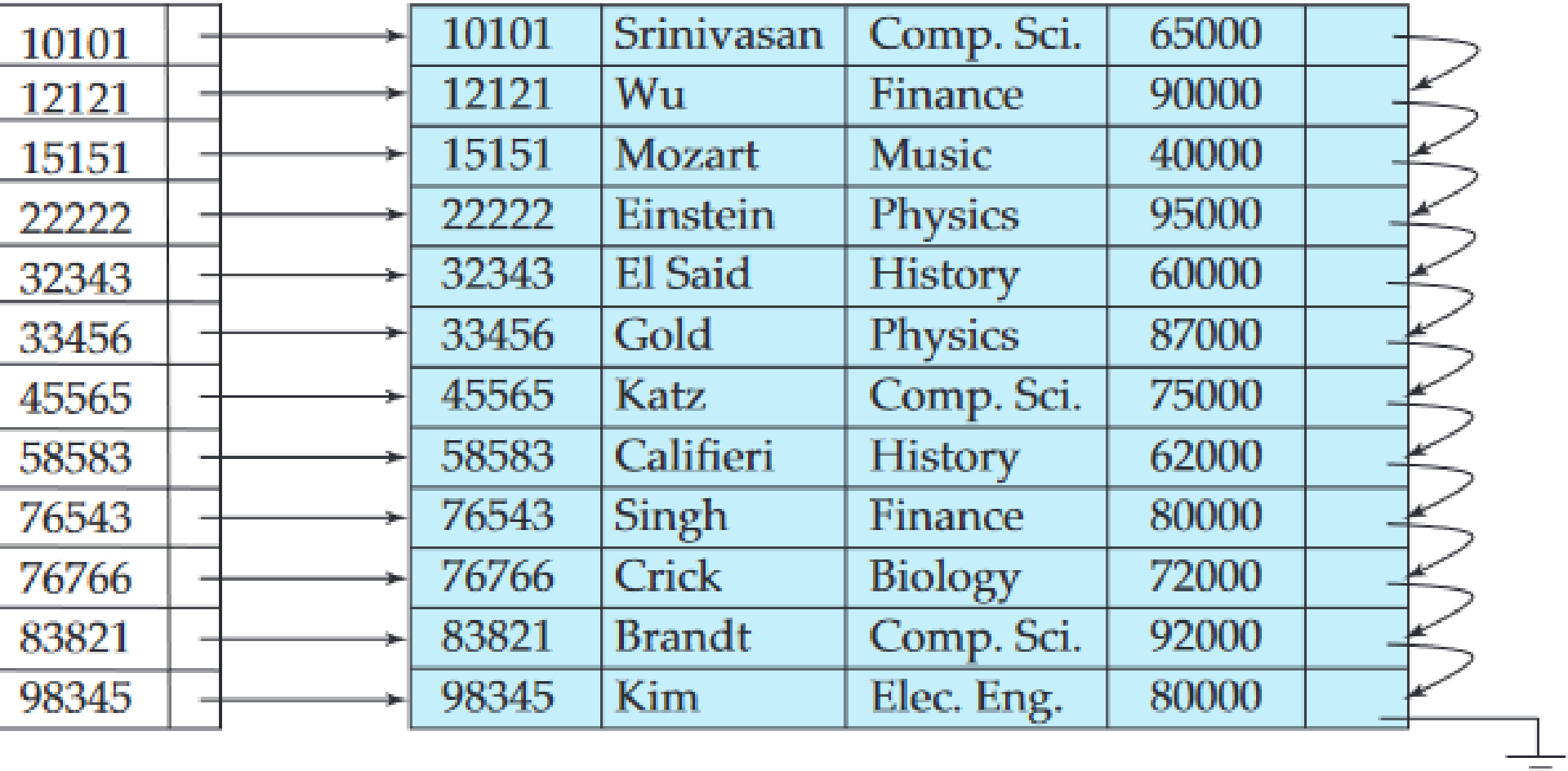


Figure 11.2 Dense index.

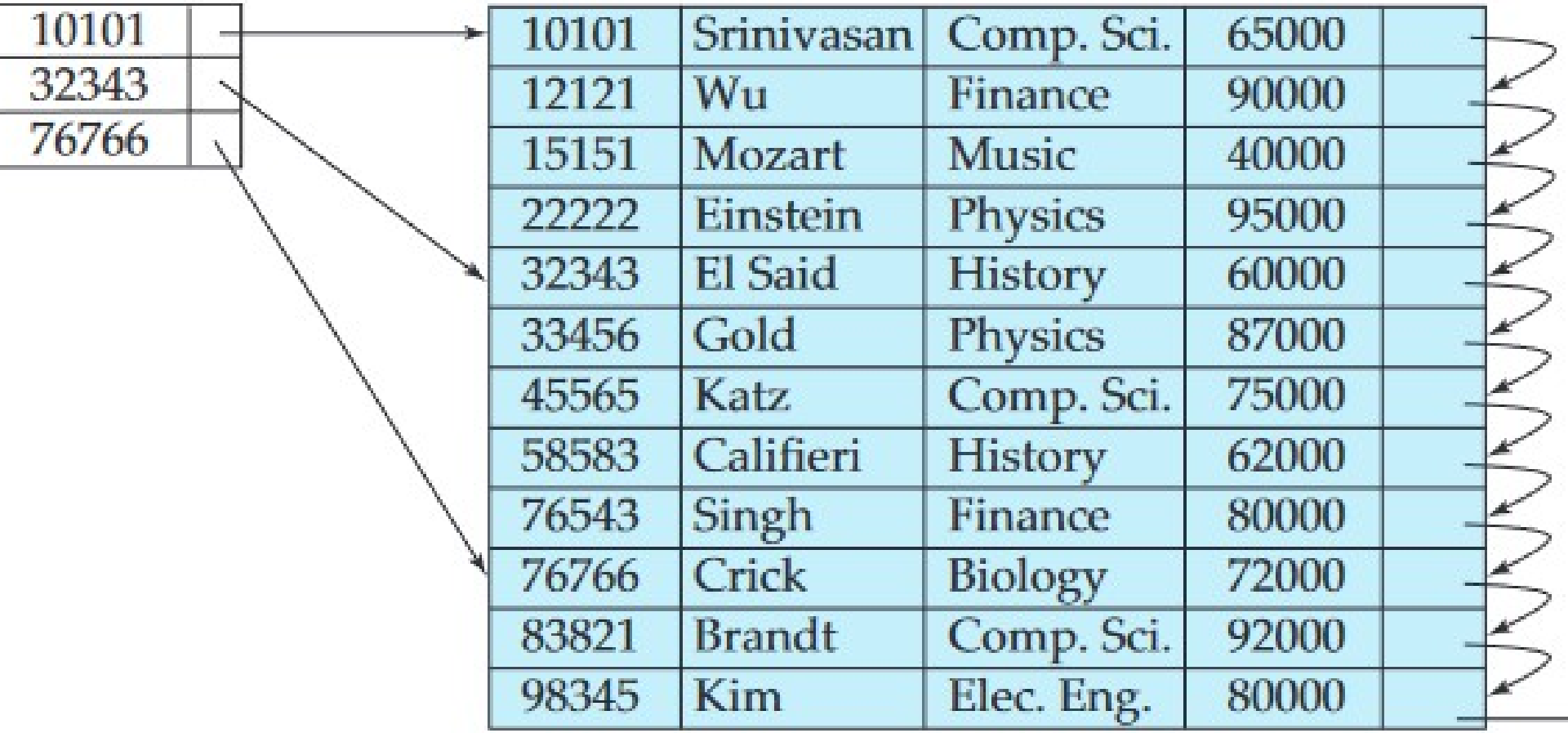
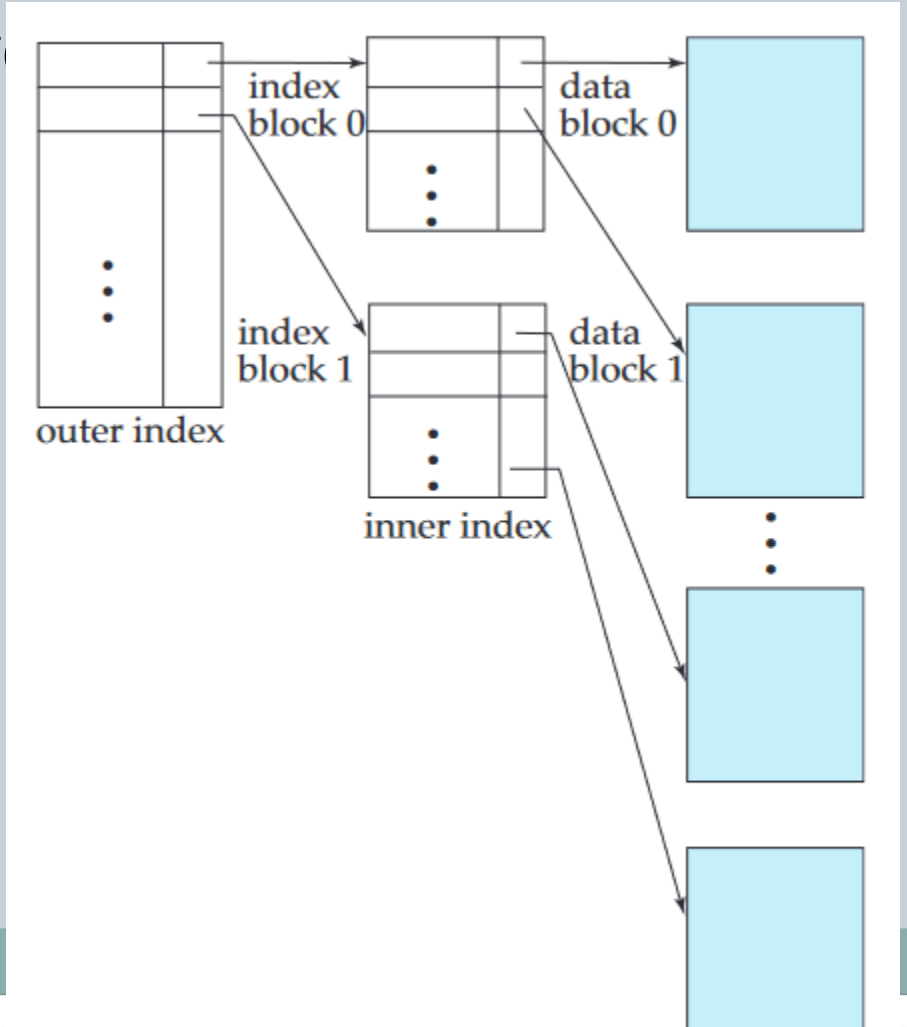


Figure 11.3 Sparse index.

multilevel indices

- Indices with two or more levels are called multilev



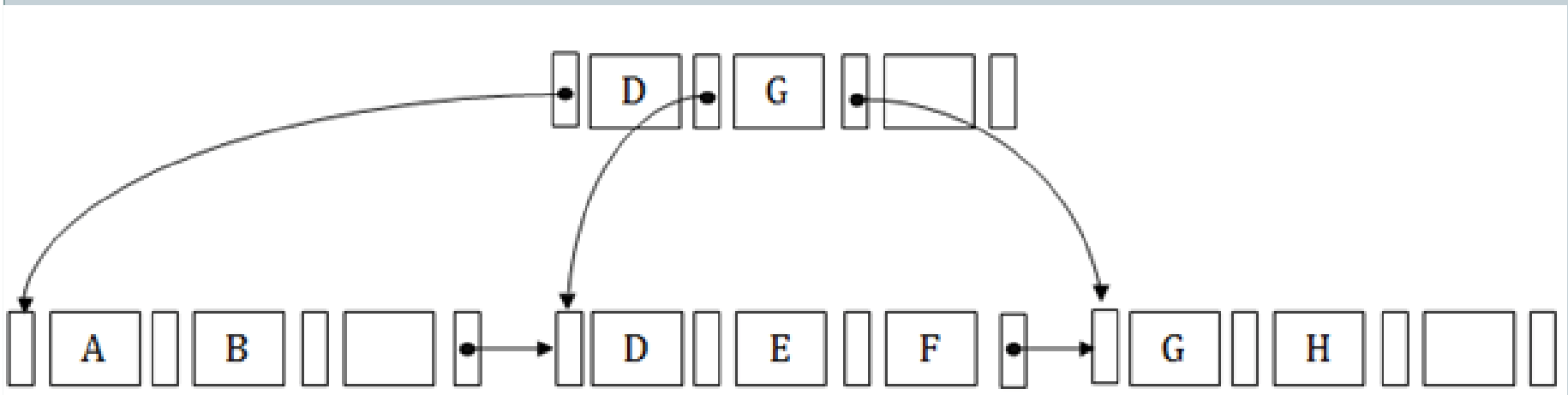
B+-Tree Index Files

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

- The main disadvantage of the index-sequential file organization is that performance degrades as the file grows.
- The **B+-tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.
- A **B+-tree** index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between [n/2] and n children, where n is fixed for a particular tree.

Structure of a B+-Tree

- In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.
- It contains an internal node and leaf node.



- A B +-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file.

Figure 11.7 Typical node of a B^+-tree.

- It contains up to n − 1 search-key values K1,K2,...,Kn − 1, and n pointers P1,P2,...,Pn. The search-key values within a node are kept in sorted order; thus, if i <j , then Ki <K j

- We consider first the structure of the leaf nodes.
- For i = 1,2,...,n−1, pointer Pi points to a file record with search-key value Ki . Pointer Pn has a special purpose
- Since there is a linear order on the leaves based on the search-key values that they contain, we use Pn to chain together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.

Figure 11.7 Typical node of a B^+-tree.

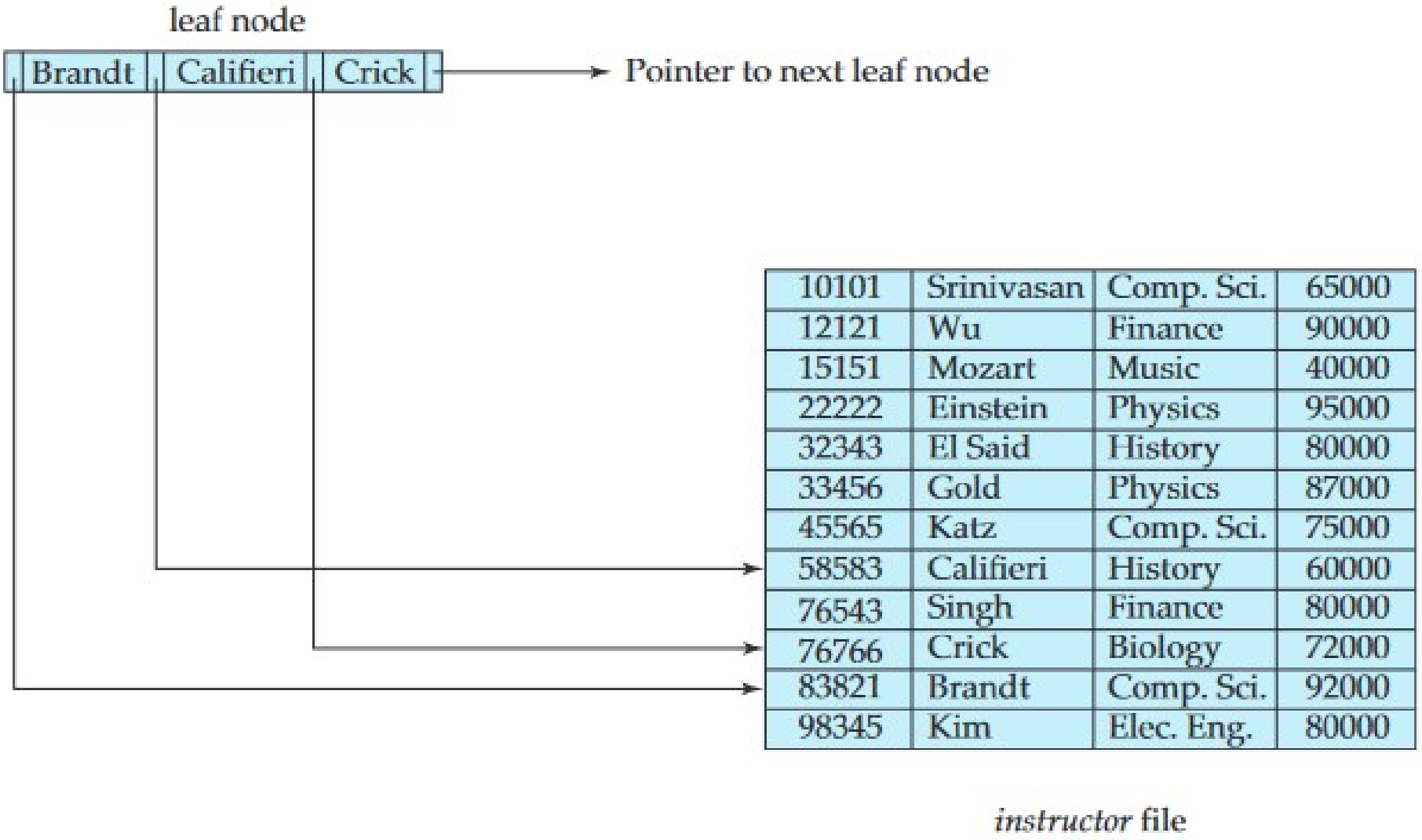


Figure 11.8 A leaf node for *instructor* B^+-tree index ($n = 4$).

- The nonleaf nodes of the B+-tree form a multilevel (sparse) index on the leaf nodes.
- The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes.
- A nonleaf node may hold up to n pointers, and must hold at least [n/2] pointers.
- The number of pointers in a node is called the fanout of the node.
- Nonleaf nodes are also referred to as internal nodes.

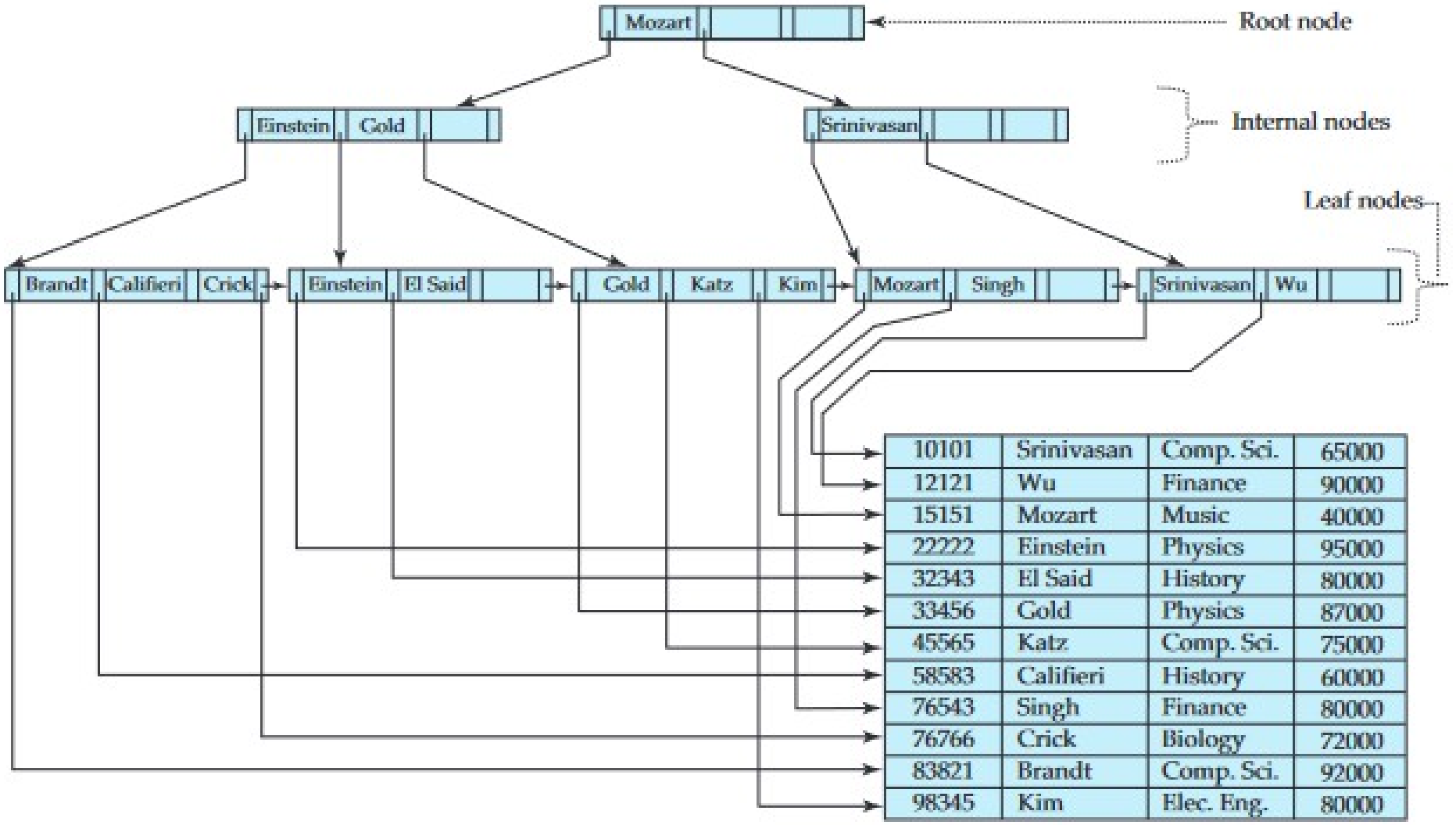


Figure 11.9 B$^+$-tree for *instructor* file ($n = 4$).

- These examples of B+-trees are all balanced. That is, the length of every path from the root to a leaf node is the same.
- This property is a requirement for a B+tree. Indeed, the “B”in B+-tree stands for “balanced.”
- It is the balance property of B+-trees that ensures good performance for lookup, insertion, and deletion.

Module4

PART-2

Hashing –Statuc hashing

- One disadvantage of sequential file organization is that we must access an index structure to locate data.
- File organizations based on the technique of hashing allow us to avoid accessing an index structure.
- In our description of hashing, we shall use 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

- let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A hash function h is a function from K to B. Let h denote
  a hash function.
- To insert a record with search key Ki , we compute h(Ki), which gives the address of the bucket for that record.
- To perform a lookup on a search-key value Ki , we simply compute h(Ki), then search the bucket with that address.

- Suppose that two search keys, K5 and K7, have the same hash value; that is, h(K5) = h(K7).
- If we perform a lookup on K5, the bucket h(K5) contains records with search-key values K5 and records with search-key values K7.
- Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want.

- Deletion is equally straightforward. If the search-key value of the record to be deleted is Ki , we compute h(Ki), then search the corresponding bucket for that record, and delete the record from the bucket.

- Hashing can be used for two different purposes. In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record.
- In a **hash index organization** we organize the search keys, with their associated pointers, into a hash file structure.

Hash Functions

- An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.
- Since we do not know at design time precisely which search-key values will be stored in the file, we want to choose a hash function that assigns search-key
  values to buckets in such a way that the distribution has these qualities:
  - The distribution is uniform.
  - The distribution is random

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

- Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file

Handling of Bucket Overflows

- If the bucket does not have enough space, a bucket overflow is said to occur. Bucket overflow can occur for several reasons:
- **Insufficient buckets**.
- The number of buckets, which we denote nB , must be chosen such that nB >nr /fr , where nr denotes the total number of records that will be stored and fr denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.

- **Skew**. Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space.
- This situation is called bucket skew.
- Skew can occur for two reasons:
  1. Multiple records may have the same search key.
  2. The chosen hash function may result in nonuniform distribution of search keys

- So that the probability of bucket overflow is reduced, the number of buckets is chosen to be (nr /fr) ∗ (1 + d), where d is a fudge factor, typically around 0.2.
  Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.

- Despite allocation of a few more buckets than required, bucket overflow can still occur.
- We handle bucket overflow by using **overflow buckets**. If a record must be inserted into a bucket b, and b is already full, the system provides an overflow bucket for b, and inserts the record into the overflow bucket.
- If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list,

- Overflow handling using such a linked list is called **overflow chaining.**

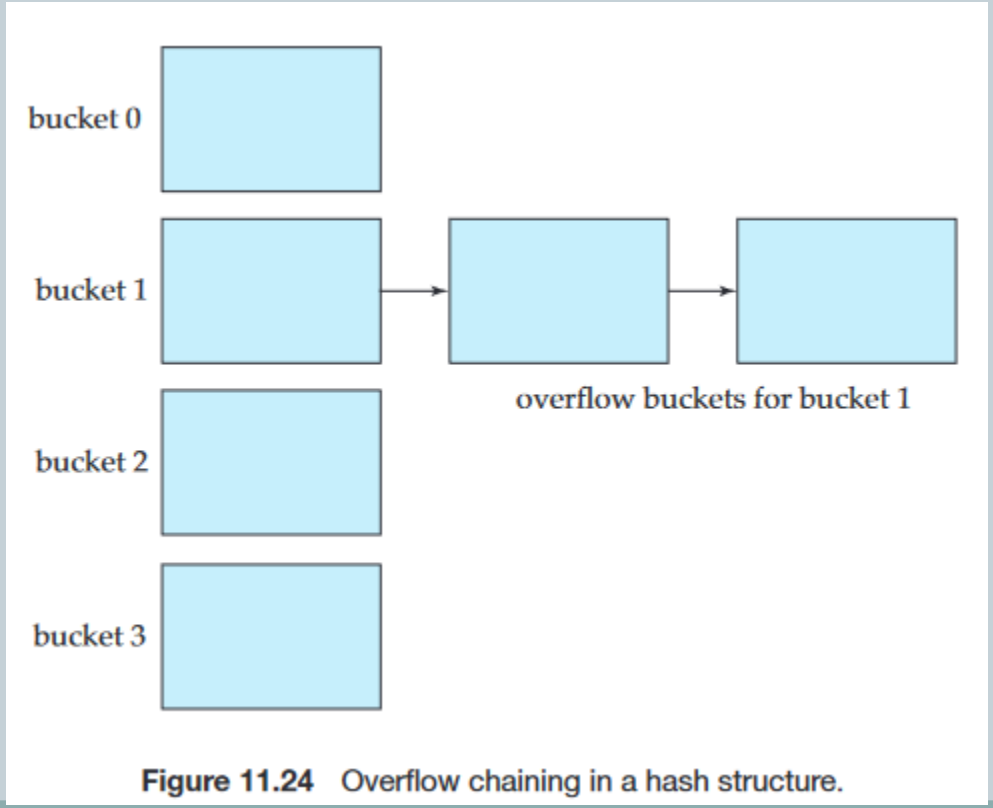


Figure 11.24 Overflow chaining in a hash structure.

- We must change the lookup algorithm slightly to handle overflow chaining.
  As before, the system uses the hash function on the search key to identify a bucket b. The system must examine all the records in bucket b to see whether they match the search key, as before. In addition, if bucket b has overflow buckets, the system
  must examine the records in all the overflow buckets also.
- The form of hash structure is closed hashing.

- Under an alternative approach, called open hashing, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets B.
  One policy is to use the next bucket (in cyclic order) that has space; this policy is called **linear probing**.

Dynamic Hashing

- the need to fix the set B of bucket addresses presents a serious problem with the static hashing technique.
- Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:
- 1. Choose a hash function based on the current file size
- 2. Choose a hash function based on the anticipated size of the file at some point in the future
- 3. Periodically reorganize the hash structure in response to file growth.

- Several dynamic hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database.
- Extendable hashing(dynamic hashing) copes with changes in database size by splitting and combining buckets as the database grows and shrinks. As a result, space efficiency is retained.
- With extendable hashing, we choose a hash function h with the desirable properties of uniformity and randomness. However, this hash function generates values over a relatively large range—namely, b-bit binary integers. A typical value for b is 32.