# Chapter 7. Data Storage and Indexing

## CHAPTER 7

## DATA STORAGE AND INDEXING

***After reading this chapter, the reader will understand:***

- *Different types of storage devices including primary storage, secondary storage, and tertiary storage*
- *Important characteristics of storage media*
- *Magnetic disk and its organization*
- *Seek time, rotational delay, and data transfer time*
- *Redundant array of independent disk, that is, RAID*
- *Performance and reliability improvement of disk with RAID*
- *All the RAID levels from level 0 to 6*
- *New storage systems*
- *The concept of files and pages*
- *Responsibilities of buffer manager*
- *Management of buffer space*
- *Page-replacement policies*
- *Fixed-length and variable-length records*
- *Spanned and unspanned organization of records in disk blocks*
- *Various file organizations including heap file organization, sequential file organization, and hash file organization*
- *Hashing (both static and dynamic), hash function, and hash table*
- *Collision and collision resolution*
- *The concept of bucket*
- *Primary index, clustering index, and secondary index*
- *Multilevel indexes including B-tree and $B^+$-tree*
- *Indexes on multiple keys*

In the previous chapters, we have viewed the database as a collection of tables (relational model) at the conceptual or logical level. The database users are largely concerned with the logical level of the database as they have hardly to do anything with the physical details of the implementation of the database system. The database is actually stored on a storage medium in the form of files, which is a collection of records consisting of related data items such as name, address, phone, email-id, and so on.

In this chapter, we start our discussion with the characteristics of the underlying storage medium. There are various media available to store data; however, database is typically stored on the magnetic disk because of its higher capacity and persistent storage. This chapter also discusses a number of methods used to organize files such as heap, sequential, index sequential, direct, etc., and the issues involved in the choice of a method. Further, the goal of a database system is to simplify and facilitate efficient access to data. One such technique is indexing, which is also discussed in this chapter.

## 7.1 HIERARCHY OF STORAGE DEVICES

In a computer system, several types of storage media such as cache memory, main memory, magnetic disk, etc. exist. On the basis of their characteristics such as cost per unit of data and speed with which data can be accessed, they can be arranged in a hierarchical order (see Figure 7.1). In addition to the speed and cost of the various storage media, another issue is volatility. **Volatile storage** loses its contents when power supply is switched off, whereas **non-volatile storage** does not.

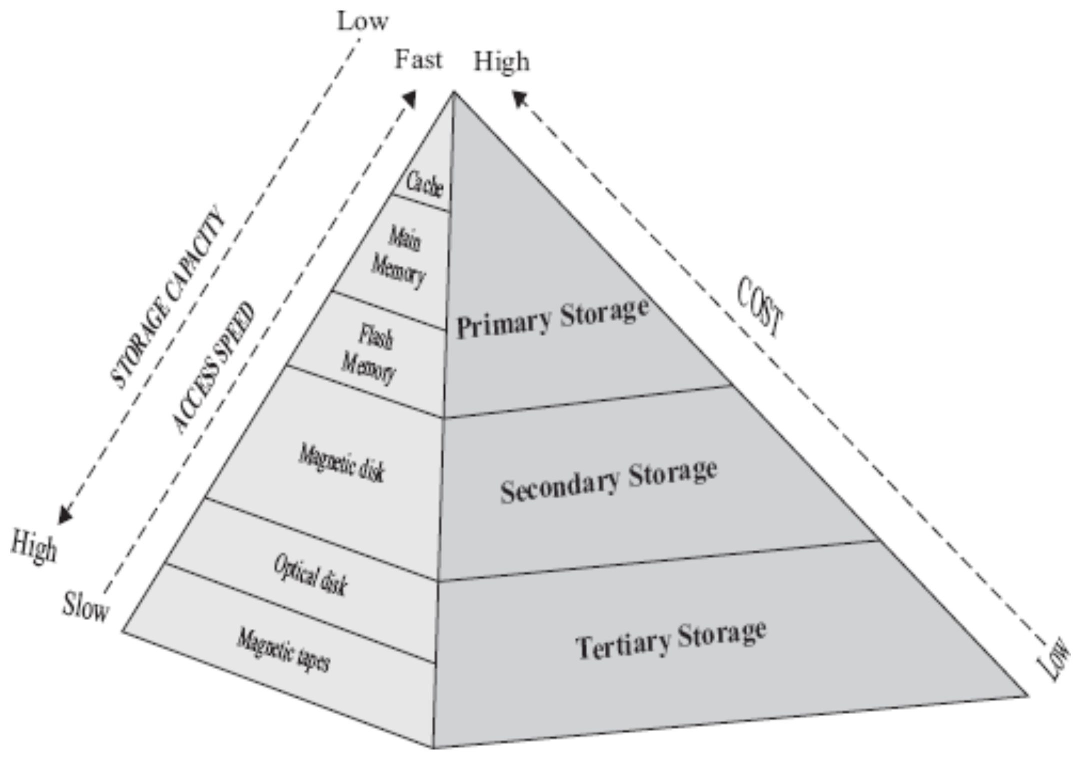


**Fig. 7.1** *Storage device hierarchy*

The cost per unit of data as well as the speed of accessing data decreases while moving down in the memory hierarchy. The storage media at the top such as cache and main memory is the highest speed memory and is referred to as **primary storage**. The storage media in the next level consists of slower devices, such as magnetic disk, and is referred to as **secondary storage**. The storage media in the lowest level is the slowest storage devices, such as optical disk and magnetic tape, and are referred to as **tertiary storage**. Let us discuss these storage devices in detail.

***Primary Storage***

Primary storage generally offers limited storage capacity due to its high cost but provides very fast access to data. Primary storage is usually made up of semiconductor chips. Semiconductor memory may be volatile or non-volatile in nature. The two basic forms of semiconductor memory are static RAM or SRAM (used for cache memory) and dynamic RAM or DRAM (used for main memory).

- **Cache memory:** Cache memory is a static RAM and is very small in size. The cache memory is the fastest; however, most expensive among all the storage media available. It is directly accessed by the CPU and is basically used to speed up the execution. However, major limitation of cache is high cost and its volatility.

- **Main memory:** Main memory is a dynamic RAM (DRAM) and is used to keep the data that is currently being accessed by the CPU. Its storage capacity typically ranges from 64 MB to 512 MB or sometimes it is extended up to hundreds of gigabytes. It offers fast access to data as compared to other storage media except static RAM. However, it is expensive and offers limited storage capacity. Thus, entire database cannot be placed in main memory at once. Moreover, it is volatile in nature.

Another form of semiconductor memory is **flash memory**. Flash memory uses an electrical erasing technology (like EEPROM). Reading data from flash memory is as fast as reading it from main memory. However, it requires an entire block to be erased and written at once. This makes the writing process little complicated.

Flash memory is non-volatile in nature and the data survives system crash and power failure. This type of memory is commonly used for storing data in various small appliances such as digital cameras, MP3 players, USB (Universal Serial Bus) storage accessories, etc.

### ***Secondary Storage***

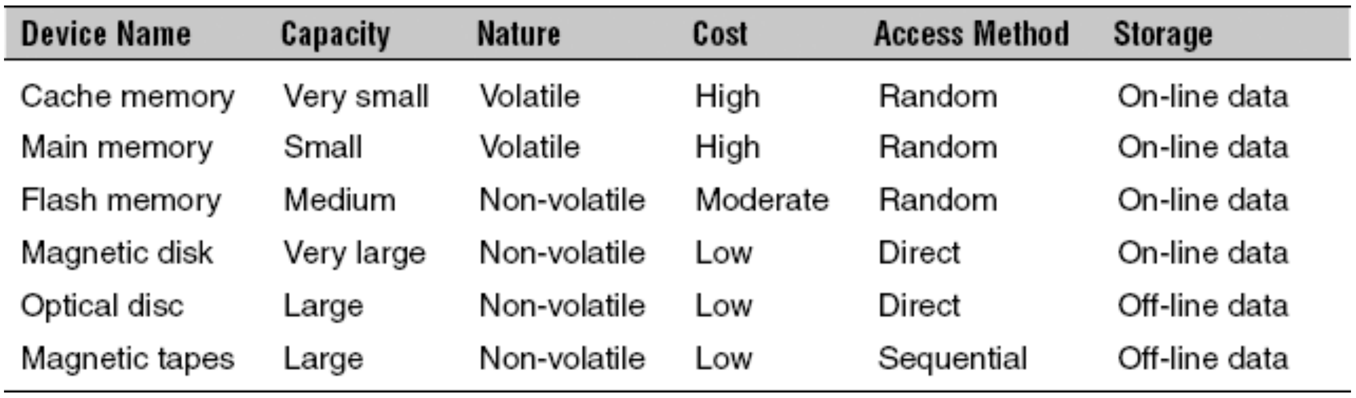
Secondary storage devices usually provide bulk of storage capacity for storing data but are slower than primary storage devices. Secondary storage is non-volatile in nature, that is, the data is permanently stored and survives power failure and system crashes. Data on the secondary storage are not directly accessed by the CPU. Instead, data to be accessed must move to main memory so that it can be accessed. **Magnetic disk** (generally called disk) is the primary form of secondary storage that enables storage of enormous amount of data. It is used to hold on-line data for a long term. Generally, the entire database is placed on magnetic disk. Magnetic disk is discussed in detail in Section 7.1.1.

### *Tertiary Storage*

Removable media, such as optical discs and magnetic tapes, are considered as tertiary storage. Larger capacity and least cost are major advantage of tertiary storage. Data access speed; however, is much slower than primary storage. Since these devices can be removed from the drive, they are also termed as **off-line storage**.

- **Optical disc:** Optical disc comes in various sizes and capacities. A compact disc—read only (CD-ROM) with a capacity of 600–700 MB of data having 12 cm diameter is the most popular means of optical disc. Another most common type of optical disc with high-storage capacity is the DVD-ROM, which stands for digital video disc—read only memory. It can store up to 8.5 GB of data per side and DVD allows for double-sided discs. As the name suggests, both the CD-ROM and DVD-ROM come pre-recorded with data, which cannot be altered. However, both use laser beams for performing read operations.
- **Tape storage:** Tape storage is cheap and reliable storage medium for organizing archives and taking backup. However, tape storage is not suitable for data files that need to be revised or updated often because it stores data in a sequential manner and offers only **sequential-access** to the stored data. Tape now has a limited role because disk has proved to be a superior storage medium. Furthermore, disk data allows **direct-access** to the stored data. Table 7.1 lists some of the important characteristics of various storage media available in computer system.

**Table 7.1** *Important characteristics of storage media*



| Device Name | Capacity | Nature | Cost | Access Method | Storage |
|---|---|---|---|---|---|
| Cache memory | Very small | Volatile | High | Random | On-line data |
| Main memory | Small | Volatile | High | Random | On-line data |
| Flash memory | Medium | Non-volatile | Moderate | Random | On-line data |
| Magnetic disk | Very large | Non-volatile | Low | Direct | On-line data |
| Optical disc | Large | Non-volatile | Low | Direct | Off-line data |
| Magnetic tapes | Large | Non-volatile | Low | Sequential | Off-line data |

### 7.1.1 Magnetic Disks

A magnetic disk is the most commonly used secondary storage medium. It offers high storage capacity and reliability. It can easily accommodate entire database at once. However, if the data stored on the disk needs to be accessed by CPU it is first moved to the main memory and then the required operation is performed. Once the operation is performed, the modified data must be copied back to the disk for consistency of the database. The system is responsible for transferring the data between disk and the main memory as and when required. Data on the disk survives power failures and system crash. There is a chance that disk may sometimes fail itself and destroy the data; however, such failures occur rarely.

Data is represented as magnetized spots on a disk. A magnetized spot represents a 1 (bit) and the absence of a magnetized spot represents a 0 (bit). To read the data, the magnetized data on the disk is converted into electrical impulses, which is transferred to the processor. Writing data onto the disk is accomplished by converting the electrical impulses from the processor into magnetic spots on the disk. The data in a magnetic disk can be erased and reused virtually infinitely. The disk is designed to reside in a protective case or cartridge to shield it from the dust and other external interference.

#### *Organization of Magnetic Disks*

A magnetic disk consists of **plate** / **platter**, which is made up of metal or glass material, and its surface is covered with magnetic material to store data on its surface. If the data can be stored on only one side of the platter, the disk is **single-sided disk**, and if both sides are used to hold the data, the disk is **double-sided disk**. When the disk is in use, the spindle motor rotates the platters at a constant high speed. Usually the speed at which they rotate is 60, 90, or 120 revolutions per second.

Disk surface of a platter is divided into imaginary tracks and sectors. **Tracks** are concentric circles where the data is stored, and are numbered from the outermost to the innermost ring, starting with zero. There are about 50,000 to 100,000 tracks per platter and a disk generally has 1 to 5 platters. Tracks are further subdivided into sectors (or track sector). A **sector** is just like an arc that forms an angle at the center. It is the smallest unit of information that can be transferred to/from the disk. There are about hundreds of sectors per track and the sector size is typically 512 bytes. The inner tracks are of smaller length than the outer tracks, thus, there are about 500 sectors per track in the inner tracks, and about 1000 sectors per track towards the boundary. In general, the disk containing large number of tracks on each surface of platter and more sectors per track has higher storage capacity.

A disk contains one **read-write head** for each surface of a platter, which is used to store and retrieve data from the surface of platter. Information is magnetically stored on a sector by the read-write head. The head moves across the surface of platter to access different tracks. All the heads are attached to a single assembly called a **disk arm**. Thus, all the heads of different platters move together. The disk platters mounted on a **spindle** together with the heads mounted on a disk arm is known as **head-disk assemblies**. All the read-write heads are on the equal diameter track on different platters at one time. The tracks of equal diameter on different platters form a **cylinder**. Accessing data of one cylinder is much faster than accessing data that is distributed among different cylinders. A close look at the internal structure of magnetic disk is shown in Figure 7.2.

**Learn More**

Some disks have one read-write head for each track of platter. These disks are termed as **fixed-head** disks, since one head is fixed on each track and is not moveable. On the other hand, disks in which the head moves along the platter surface are termed as **moveable-head** disks.

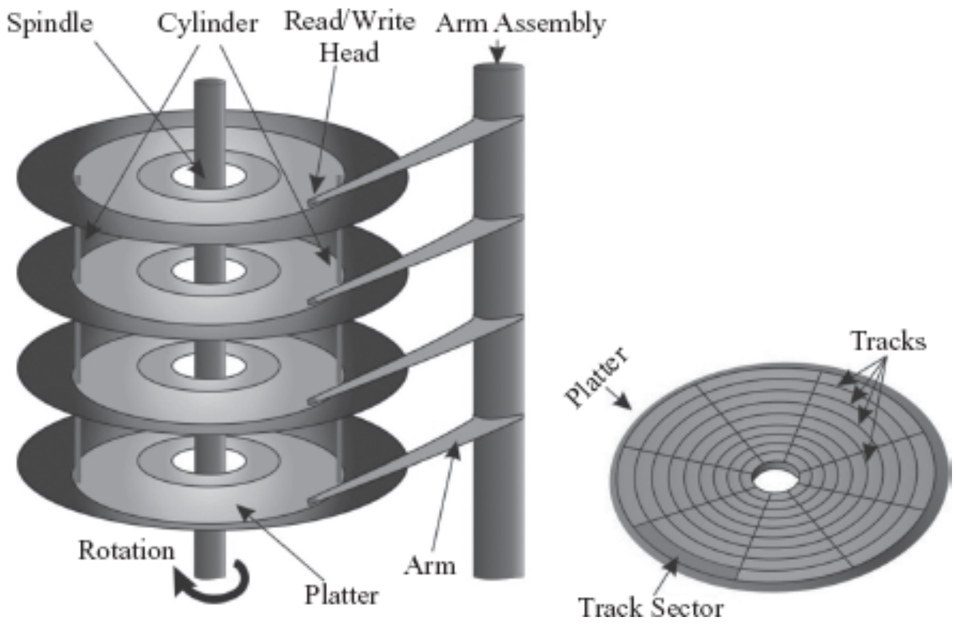


**Fig. 7.2** *Moving head disk mechanism*

***Accessing Data from Magnetic Disk***

Data in a magnetic disk is recorded on the surface of the circular tracks with the help of read/write head, which is mounted on the arm assembly. These heads can be multiple in numbers to access the adjacent tracks simultaneously and thus access to a disk is faster. The transfer of data between memory and disk drive is handled by a **disk controller**, which interfaces the disk drive to the computer system.

Some common interfaces used for disk drives on personal computers and workstations are **small-computer-system interconnect** (**SCSI)** (pronounced "scuzzy"), **AT attachment** (**ATA**), and **serial ATA (SATA)**. In the latest technology, disk controller is implemented within the disk drive. The controller accepts high level I/O commands (to read or write sector) and start positioning the disk arm over the right track in order to read or write the data. Disk controller computes the **checksums** for the data to be written on the sector and attach it with the sector. When the sector is to be read, the controller again computes the checksum from the sector data and compares it with stored checksum. If there is any difference between them, the controller will retry reading data several times. However, if the difference between computed and stored checksum continues to occur, the controller signals a read failure.

**Remapping of bad sectors** is another major task performed by the disk controllers. During the initial formatting of disk, if the controller detects a bad (or damaged) sector, it is logically mapped to another physical location by the disk controller. Disk is notified for the remapping and any further operation is carried out on the new location.

The process of accessing data comprises three steps:

1. **Seek:** As soon as the disk unit receives the read/write command, the read/write heads are positioned on specific track on the disk platter. The time taken in doing so is known as **seek time**. It is average time required to move the heads from one track to some other desired track on the disk. Seek times of modern disk may range between 6–15 milliseconds.
2. **Rotate:** Once the heads are positioned on the desired track, the head of the specific platter is activated. Since the disk is rotated constantly, the head has to wait for the required sector or cluster (desired data) to come under it. This delay is known as **rotational delay time** or **latency** of the disk. The average rotational latencies range from 4.2 to 6.7 ms.
3. **Data transfer:** After waiting for the desired data location, the read/write head transfers the data to or from the disk to primary memory. The rate at which the data is read from or written to the disk is known as **data transfer rate**. It is measured in kilobits per second (kbps). Some of the latest hard disks have a data transfer rate of 66 MB/second. The data transfer rate depends upon the rotational speed of the disk. If the disk has a rotational speed of 6000 rpm (rotations per minute), having 125 sectors and 512 bytes/sector, the data transfer rate per revolution will be 125 × 512 = 64000 bytes. Hence, the total transfer rate per second will be 64000 × 6000/60 = 6,400,000 bytes/ second or 6.4 MB/second.

The combined time (seek time, latency time, and data transfer time) is known as the **access time**. Specifically, it can be described as the period of time that elapses between a request for information from disk or memory, and the information arriving at the requesting device. **Memory access time** refers to the time it takes to transfer a character from memory to or from the processor, while **disk access time** refers to the time it takes to place the read/write heads over the given sector and transfer the data to or from the requested device. RAM may have an access time of 80 nanoseconds or less, while disk access time could be 12–19 milliseconds.

The **reliability** of the disk is measured in terms of **mean time to failure (MTTF)**. It is the amount of time for which the system can run continuously without any failure. Manufacturers claim that the mean time to failure of disks ranges between 1,000,000 hours (about 116 years) to 1,500,000 hours (about 174 years). Although, various research studies conclude that failure rates are, in some cases, 13 times greater than what manufacturer claims. Generally, expected life span of most disks is about 4 to 5 years. However, disks have a high rate of failure when they become a few years old.

## 7.2 REDUNDANT ARRAYS OF INDEPENDENT DISKS

The technology of semiconductor memory has been advancing at a much higher rate than the technology of secondary storage. The performance and capacity of semiconductor memory is much superior to secondary storage. To match this growth in semiconductor memory, a significant development is required in the technology of secondary storage. A major advancement in secondary storage technology is represented by the development of **Redundant Arrays of Independent Disks (RAID)**. The basic idea behind RAID is to have a large array of small independent disks. Presence of multiple disks in the system improves the overall transfer rates, if the disks are operated in parallel. Parallelizing the operation of multiple disks allow multiple I/O to be serviced in parallel. This setup also offers opportunities for improving the reliability of data storage, because data can be stored redundantly on multiple disks. Thus, failure of one disk does not lead to loss of data. In other words, this large array of independent disks acts as a single logical disk with improved performance and reliability.

***Improving Performance and Reliability with RAID***

In order to improve the performance of disk, a concept called **data striping** is used which utilizes parallelism. Data striping distributes the data transparently among `N` disks, which make them appear as a single large, fast disk. Striping of data across multiple disks improves the transfer rate as well, since operations are carried out in parallel. Data striping also balances the load among multiple disks.

**Learn More**

Originally RAID stands for Redundant Array of Inexpensive Disk, since array of cheap smaller capacity disk was used as an alternative to large expensive disk. Those days the cost per bit of data of smaller disk was less than that of larger disk.

In the simplest form, data striping splits each byte of data into bits and stores them across different disks. This splitting of each byte into bits is known as **bit-level data striping**. Having 8-bits per byte, an array of eight disks (or either a factor or multiple of eight) is treated as one large logical disk. In general, bit `i` of each byte is written $i^{th}$ disk. However, if an array of only two disks is used, all odd numbered bits go to first disk and even numbered bits to second disk. Since each I/O request is accomplished with the use of all disks in the array, transfer rate of I/O requests goes to `N` times, where N represents the number of disks in the array.

Alternatively, blocks of a file can be striped across multiple disks. This is known as **block-level striping.** Logical blocks of a file are assigned to multiple disks. Large requests for accessing multiple blocks can be carried out in parallel, thus improving data transfer rate. However, transfer rate for the request of a single block is same as it is in case of one disk. Note that the disks that are not participating in the request are free to carry out other operations.

Having an array of `N` disks in a system improves the system performance; however, lowers the overall storage system reliability. The chance of failure of at least one disk out of total `N` disks is much higher than that of a specific single disk. Assume that the mean time to failure (MTTF) of a disk is about 150,000 hours (slightly over 16 years). Then, for an array of 100 disks, the MTTF of some disk is only 150,000/100 = 1500 hours (about 62 days). With such short MTTF of a disk, maintaining one copy of data in an array of `N` disks might result in loss of significant information. Thus, some solution must be employed to increase the reliability of such storage system. The most acceptable solution is to have **redundant** information. Normally, the redundant information is not needed; however, in case of disk failure it can be used to restore the lost information of failed disk.

One simple technique to keep redundant information is **mirroring** (also termed as **shadowing**). In this technique, the data is redundantly stored on two physical disks. In this way every disk is duplicated, and all the data has two copies. Thus, every write operation is carried on both the disks. During read operation, the data can be retrieved from any disk. In case of failure of one disk, second disk can be used until the first disk gets repaired. If the second disk also fails before the repairing of first disk is completed, the data is lost. However, occurrence of such event is very rare. The mean time to failure of mirrored disk depends on two factors.

1. **Mean time to failure** of independent disks and
2. **Mean time to repair** a disk. It is the time taken, on an average, to restore the failed disk or to replace it, if required.

Suppose failure of two disks is independent of each other. Further, assume that the mean time to repair of a disk is 15 hours and mean time to failure of single disk is 150,000 hours, then mean time to data loss in mirrored disk system is $(150{,}000)^2 / (2 * 15) = 7.5 * 10^8$ hours or about 85616 years.

An alternative solution to increase the reliability is storing error-correcting codes such as parity bits and hamming codes (discussed in next section). Such additional information is needed only in case of recovering the data of failed disk. Error-correcting codes are maintained in a separate disk called **check disk.** The parity bits corresponding to each bit of `N` disks are stored in check disk.

### 7.2.1 RAID Levels

Data striping and mirroring techniques improve the performance and reliability of a disk. However, mirroring is expensive and striping does not improve reliability. Thus, several RAID organizations referred to as **RAID levels** have been proposed which aim at providing redundancy at lower cost by using the combination of these two techniques. These levels have different cost-performance trade-offs. The RAID levels are classified into seven levels (from level 0 to level 6), as shown in Figure 7.3, and are discussed here. To understand all the RAID levels consider a disk array consisting of 4 disks.

- **RAID level 0:** RAID level 0 uses block-level data striping but does not maintain any redundant information. Thus, the write operation has the best performance with level 0, as only one copy of data is maintained and no redundant information needs to be updated. However, RAID level 0 does not have the best read performance among all the RAID levels, since systems with redundant information can schedule disk access based on shortest expected seek time and rotational delay. In addition, RAID level 0 is not fault-tolerant because failure of just one drive will result in loss of data. However, absence of redundant information ensures 100 per cent space utilization for RAID level 0 systems.
- **RAID level 1:** RAID level 1 is the most expensive system as this level maintains duplicate or redundant copy of data using mirroring. Thus, two identical copies of the data are maintained on two different

disks. Every write operation needs to update both the disks, thus, the performance of RAID level 1 system degrades while writing. However, performance while read operation is improved by scheduling request to the disk with the shortest expected access time and rotational delay. With two identical copies of data, RAID level 1 system ensures only 50 per cent space utilization.

- **RAID level 2:** RAID level 2 is known as **error-correcting code** (**ECC**) organization. Two most popular error detecting and correcting codes are parity bits and hamming codes. In memory system, each byte is associated with a parity bit. The parity bit is set to 0 if the number of bits in the byte that are set to 1 is even; otherwise the parity bit is set to 1. If any one bit in the byte gets changed, then parity of that byte will not match with the stored parity bit. In this way, use of parity bit detects all 1-bit errors in the memory system. Hamming code has the ability to detect the damaged bit. It stores two or more extra bits to find the damaged bit and by complementing the value of damaged bit, the original data can be recovered. RAID level 2 requires three redundant disks to store error detecting and correcting information for four original disks. Thus, effective space utilization is about 57 per cent in this case. However, space utilization increases with the number of data disks because check disks grow logarithmically with the number of data disks.
- **RAID level 3:** As discussed, RAID level 2 uses check disks to hold information to detect the failed disk. However, disk controllers can easily detect the failed disk and hence, check disks need not to contain information to detect the failed disk. RAID level 3 maintains only single check disk with parity bit information for error correction as well as for detection. This level is also named as **bit-interleaved parity organization**.

  Performance of RAID level 2 and RAID level 3 are very similar but RAID level 3 has lowest possible overheads for reliability. So, in practice, level 2 is not used. In addition, level 3 has two benefits over level 1. Level 1 maintains one mirror disk for every disk, whereas level 3 requires only one parity disk for multiple disks, thus, increasing effective space utilization. In addition, level 3 distributes the data over multiple disks, with `N`-way striping of data, which makes the transfer rate for reading or writing a single block by `N` times faster than level 1. Since every disk has to participate in every I/O operation, RAID level 3 supports lower number of I/O operations per second than RAID level 1.

- **RAID level 4:** Like RAID level 0, RAID level 4 uses block-level striping. It maintains parity block on a separate disk for each corresponding block from `N` other disks. This level is also named as **block-interleaved parity organization**. To restore the block of the failed disk, blocks from other disks and corresponding parity block is used. Requests to retrieve data from one block are processed with only one disk, leaving remaining disks free to handle other requests. Writing a single block involves one data disk and check disk. The parity block is required to be updated with each write operation, thus, only one write operation can be processed at a particular point of time. With four data disks, RAID level 4 requires just one check disk. Effective space utilization for our example of four data disk is 80 per cent. However, as always one check disk is required to hold parity information, effective space utilization increases with the number of data disks.
- **RAID level 5:** Instead of placing data across `N` disks and parity information in one separate disk, this level distributes the **block-interleaved** parity and data among all the `N+1` disks. Such distribution has advantage in processing read/write requests. All disks can participate in processing read request, unlike RAID level 4, where dedicated check disks never participates in read request. So level 5 can satisfy more number of read requests in given amount of time. Since bottleneck of single check disk has been eliminated, several write request could also be processed in parallel. RAID level 5 has the best performance among all the RAID levels with redundancy. In our example of 4 actual disks, RAID level 5 has five disks overall, thus, effective space utilization for level 5 is same as in

level 3 and level 4.

- **RAID level 6:** RAID level 6 is an extension of RAID level 5 and applies P + Q redundancy scheme using Reed-Solomon codes. **Reed–Solomon codes** enable RAID level 6 to recover from up to two simultaneous disk failures. RAID level 6 requires two check disks; however, like RAID level 5, redundant information is distributed across all disks using block-level striping.

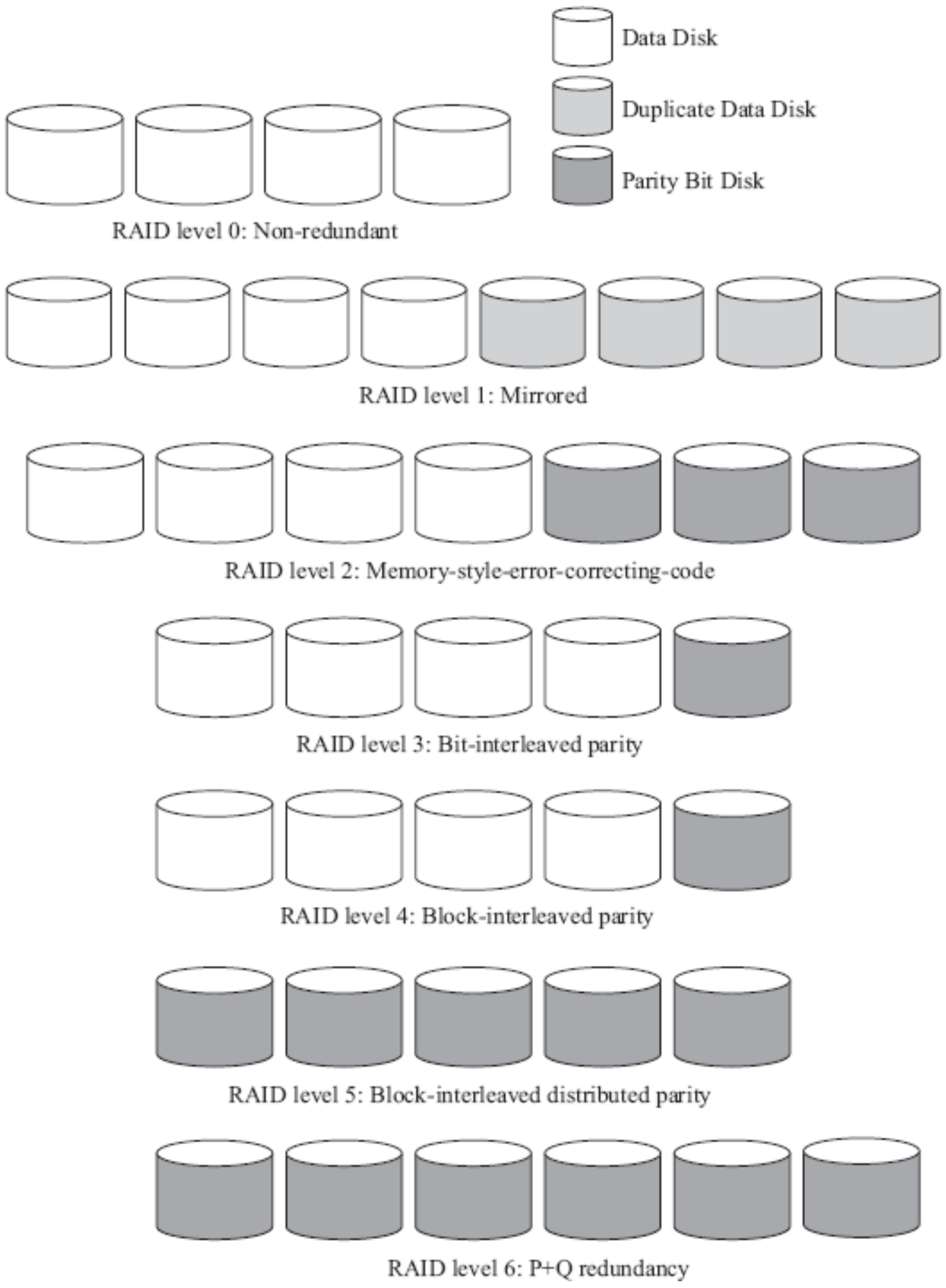

Data Disk
Duplicate Data Disk
Parity Bit Disk
RAID level 0: Non-redundant
RAID level 1: Mirrored
RAID level 2: Memory-style-error-correcting-code
RAID level 3: Bit-interleaved parity
RAID level 4: Block-interleaved parity
RAID level 5: Block-interleaved distributed parity
RAID level 6: P+Q redundancy

**Fig. 7.3** *Representing RAID levels*

## 7.3 NEW STORAGE SYSTEMS

Demand for storage of data is increasing across the organizations with the advent of Internet-driven applications such as e-commerce. Enterprise Resource Planning (ERP) systems and data warehouses need enough space to keep bulk data or information across the organization. In addition, the organizations have different branches across the world and providing data to different users across different branches in 24×7 environment is a challenging task. As a result, managing all the data becomes costly with an increase in simultaneous requests. In some cases, cost of managing server-attached storage exceeds the cost of the server itself. Therefore, various organizations choose the concept called **storage area network (SAN)** to manage their storage.

In SAN architecture, many server computers are connected with a large number of disks on a high-speed network. Storage disks are placed at a central server room, and are monitored and maintained by system administrators. The storage subsystem and the computer communicate with each other by using SCSI or fiber channel interfaces (FCI). Several SAN providers came up with their own topologies, thus, providing different performance and connectivity options allowing storage subsystem to be placed far apart from the server computers. Storage subsystem is shared among multiple computers that could process different parts of an application in parallel. Servers are connected to multiple RAID systems, tape libraries, and other storage systems in different configurations by using fiber-channel switches and fiber-channel hubs. Thus, the main advantage is many-to-many connectivity among server computers and storage system. Furthermore, isolation capability of SAN allows easy addition of new peripherals and servers.

An alternative to SAN is **network-attached storage (NAS)**. A NAS device is a server that allows file sharing. NAS does not provide any of the services that a typical server provides. NAS devices are being used for high performance storage solutions at low cost. NAS devices allow enormous amount of hard disk storage space to be made available to multiple servers without shutting them down for maintenance and upgrades. The NAS devices do not need to be located within the server, but can be placed anywhere in a local area network (LAN) and may be combined in different configurations. A single hardware device (called the **NAS box** or **NAS head**) acts as the interface between the NAS system and network clients. NAS units usually have a web interface and do not require a monitor, keyboard, or mouse. In addition, NAS system usually contains one or more hard disks or tape drives, often arranged into logical, redundant storage containers or RAID, as traditional file servers do. Further, NAS removes the file serving responsibility from other servers on the network.

NAS provides both storage and file system. It is often contrasted with SAN, which provides only block-based storage and leaves file system concern on the client side. The file based protocols used by NAS are NFS or SMB, whereas SAN protocols are SCSI or fibre channel. NAS increases the performance as the file serving is done by the NAS and not by the server, which is responsible for doing other processing. The performance of NAS devices depends heavily on the speed of and traffic on the network, and also on the amount of cache memory on the NAS devices.

## 7.4 ACCESSING DATA FROM DISK

The database is mapped into a number of different files, which are physically stored on disk for persistency and the underlying operating system is responsible for maintaining these files. Each file is decomposed into equal size **pages**, which is the unit of exchange between the disk and the main memory. The size of the page chosen is equal to the size of disk block and a page can be stored in any disk block. Hence, reading or writing a page can be done in one disk I/O. Generally, main memory is too small to hold all the pages of a file at one time. Thus, the pages of a file need to be transferred into main memory as and when requested by the CPU. If there is no free space in main memory, some existing page must be replaced to make space for the new page. The policy that is used to choose a page to be replaced with the new page is called the

**replacement policy**.

To improve the performance of a database, it is required to keep as many pages in the main memory as possible. Since it is not possible to keep all the pages in main memory at a time, the available space of main memory should be managed efficiently. The goal is to minimize the number of disk accesses by transferring the page in the main memory before a request for that page occurs in the system. The main memory space available for storing the data is called **buffer pool** and the subsystem that manages the allocation of buffer space is called **buffer manager**. The available main memory is partitioned into page size **frames**. Each frame can hold a page of file at any point of time.

### 7.4.1 Buffer Manager

When a request for a page is generated, the buffer manager handles it by passing the memory address of the frame that contains the requested page to its requester, if the page is already in the buffer pool. However, if the page is not available in buffer pool, buffer manager allocates space for that page. If no free frame is available in buffer pool, allocation of space requires de-allocation of some other pages that are no longer needed by the system. After allocating the space in buffer pool, buffer manager transfers the requested page from disk to main memory. While de-allocating a page, the buffer manager writes back the updated information of that page on the disk, if the page has been modified since its last access from the disk (see Figure 7.4).

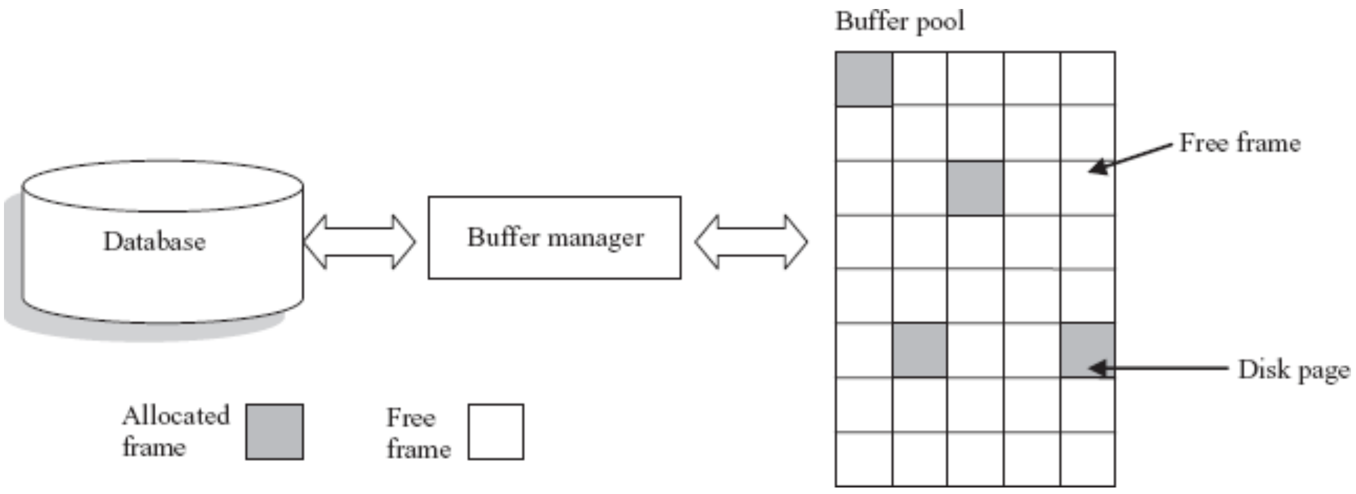


**Fig. 7.4** *Buffer management*

In addition to the buffer pool, the buffer manager maintains two variables for each frame in the buffer pool.

- **`pin_count:`** This variable keeps track of the number of times the page currently in a given frame has been requested but not released. In this way, it records the number of current users of the page.
- **`dirty:`** It is a Boolean variable that keeps track whether the current page in the frame has been modified or not, since it was brought into the frame from disk.

Initially, the value of `pin_count` is set to 0 and `dirty` bit is turned off for every frame. On receiving a request for a page, the buffer manager first searches the buffer pool for that page. If the requested page is available in any frame, its `pin_count` is incremented by one. If no frame in the buffer pool is holding the requested page, buffer manager chooses a frame for replacement using replacement policy. If the `dirty` bit is on for the replacement frame, the page in that frame is written back to the disk and then, the requested page is transferred from disk to that frame in the buffer pool. After having the requested page in buffer pool, buffer manager passes the memory address of frame that contains the requested page to its requestor.

The process of incrementing the `pin_count` is generally called **pinning** the page. When the requested page is subsequently released by the process, `pin_count` of that frame is decremented. It is called **unpinning** the page. At the time of unpinning the page, the `dirty` bit needs to be set by the buffer manager, if the page has been modified by its requestor. Note that the buffer manager will not replace the page in the frame until it is unpinned by all its requestor, that is, until its `pin_count` becomes 0.

For each new page from disk, buffer manager always choose a frame with `pin_count` 0 to be replaced, if no free frame is available. In case two or more such frames are found, then buffer manager chooses a frame according to its replacement policy.

### 7.4.2 Page-Replacement Policies

Buffer manager uses replacement policy to choose a page for replacement from the list of unpinned pages. The main goal of replacement policies is to minimize the number of disk accesses for the requested page. For general-purpose programs, it is not possible to anticipate accurately which page will be requested in near future and hence, operating system uses the pattern of past references to predict the future references.

Commonly used replacement policies include **least recently used (LRU)**, **most recently used (MRU)**, and **clock replacement**. LRU replacement policy assumes that the pages that have been referenced recently have greater chance to be referenced again in the near future. Thus, the least recently referenced page should be chosen for replacement. However, the pattern of future references is more accurately predicted by the database system than an operating system. A user-request to the database system involves several steps and by carefully looking at these steps, database system can apply this policy more efficiently than operating system.

In some cases, the optimal policy to choose a page for replacement is MRU policy. As the name indicates, MRU chooses the most recently accessed page for replacement, as and when required. Note that the page currently being used by the system must be pinned and hence, they are not eligible for replacement. After completing the processing with the page, it must be unpinned by the system so that it becomes available for replacement.

Another replacement policy is **clock replacement** policy, which is a variant of LRU. In this replacement policy, an additional `reference` bit is associated with each frame, which is set on as soon as `pin_count` of that frame becomes 0. All the frames are considered as arranged in a circular manner. A variable `current` moves around the logical circle of frames, starting with the value 1 through `N` (total number of frames in the buffer pool).

The clock replacement policy considers the `current` frame for replacement. If the considered frame is not chosen for replacement, value of `current` is incremented by one and the next frame is considered. The policy continues to consider the frame until some frame is chosen for replacement. If the `pin_count` of `current` frame is greater than 0, then the current frame is not a candidate for replacement. If the current frame has `pin_count` 0 and `reference` bit on, the algorithm turns it off—this helps to ignore the recently referenced pages to be chosen for replacement. Only the frame having `pin_count` 0 and `reference` bit off can be chosen for replacement. If in some sweep of clock, all the frames have `pin_count` greater than 0, it means no frame in the buffer pool is a candidate for replacement.

**Learn More**

Buffer manager should not attempt to choose the pages of indexes (discussed in Section 7.7) for replacement, until or unless necessary, because pages of indexes are generally accessed much frequently than the pages of the file itself.

Although, no single policy is suitable for all types of applications, a large number of database systems use LRU. The policy to be used by buffer manager is influenced by various factors other than time at which the page will be referenced again. If the database system is handling requests from several users concurrently, the system may need to delay certain requests by using some concurrency-control mechanism to maintain database consistency. Buffer manager needs to alter its replacement policy using information from concurrency control subsystem, indicating which requests are being delayed.

## 7.5 PLACING FILE RECORDS ON DISK BLOCKS

A database consists of a number of relations where each relation is typically stored as a file of records. Each record of a file represents an entity instance and its attributes. For example, each record of `PUBLISHER` relation of *Online Book* database consists of `P_ID`, `Pname`, `Address`, `State`, `Phone`, and `Email_id` fields. These file records are mapped onto disk blocks. The size of blocks are fixed and is determined by the physical properties of disk and by the operating system; however, the size of records may vary. Generally in the relational database, different relations have tuple of different sizes. Thus, before discussing how file records are mapped onto disk blocks, we need to discuss how database is represented in terms of files.

There are two approaches to organize the database in the form of files. In the first approach, all the records of a file are of fixed-length. However, in the second approach, the records of a file vary in size. A file of fixed-length records is simple to implement as all the records are of fixed-length. However, deletion of record results in fragmented memory. On the other hand, in case of files of variable-length records, memory space is efficiently utilized; however, locating the start and end of record is not simple.

### 7.5.1 Fixed-Length Records

All the records in a file of fixed-length record are of same length. Consider a relation `PUBLISHER` whose record is defined here.

| | | |
|---|---|---|
| create PUBLISHER as | | |
| (P_ID | char(5) | PRIMARY KEY NOT NULL, |
| Pname | char(20) | NOT NULL, |
| Address | char(50), | |
| State | char(20), | |
| Phone | numeric(10), | |
| Email_id | char(50)) | |

It is clear from the given definition that each record consists of `P_ID`, `Pname`, `Address`, `State`, `Phone`, and `Email_id` fields, resulting in records of fixed-length, but of varying length fields. Assume that each character requires 1 byte and numeric(10) requires 5 bytes, then a PUBLISHER record is 150 bytes long. Figure 7.5 represents a record of PUBLISHER relation with starting of each field within the record.

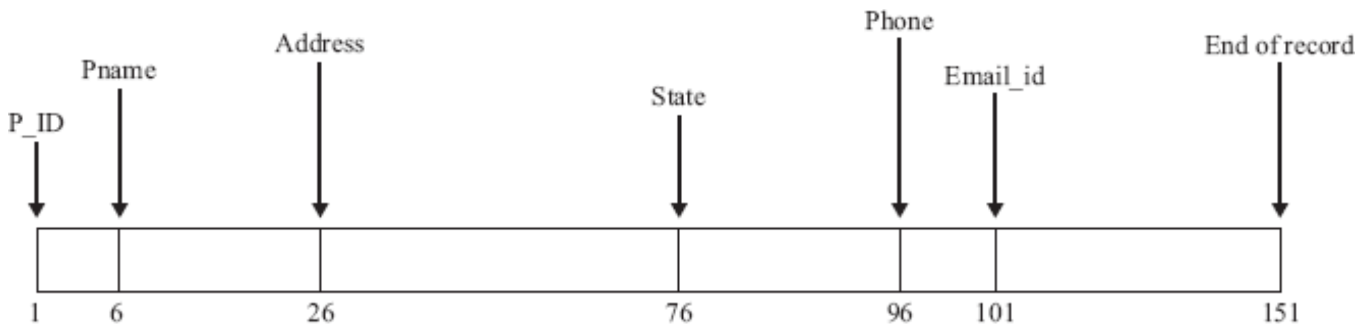


**Fig. 7.5** *Fixed-length record*

In a file of fixed-length records, every record consists of same number of fields and size of each field is fixed for every record. It ensures easy location of field values as their positions are pre-determined. Since each record occupies equal memory, as shown in Figure 7.6, identifying start and end of record is relatively simple.

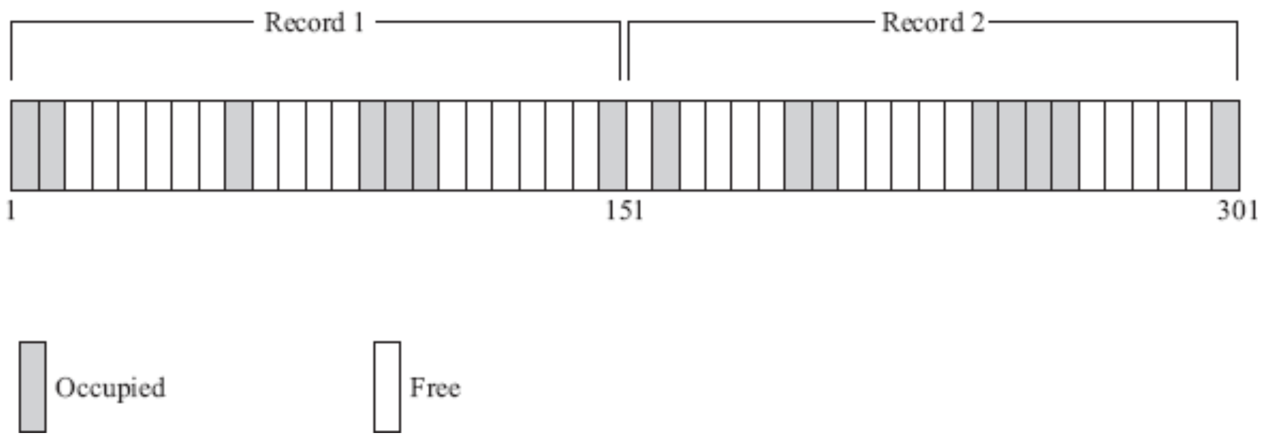


**Fig. 7.6** *Location of fixed-length records*

A major drawback of fixed-length records is that a lot of memory space is wasted. Since a record may contain some optional fields and space is reserved for optional fields as well—it stores *null* value if no value is supplied by the user for that field. Thus, if certain records do not have values for all the fields, memory space is wasted. In addition, it is difficult to delete a record as deletion of a record leaves blank space in between the two records. To fill up that blank space, all the records following the deleted record need to be shifted.

It is undesirable to shift a large number of records to fill up the space made available by a deleted record, since it requires additional disk access. Alternatively, the space can be reused by placing a new record at the time of insertion of new records, since insertions tend to be more frequent. However, there must be some way to mark the deleted records so that they can be ignored during the file scan. In addition to simple marker on deleted record, some additional structure is needed to keep track of free space created by deleted or marked records. Thus, certain number of bytes is reserved in the beginning of the file for a **file header**. The file header stores the address of first marked record, which further points to second marked record and so on. As a result, a linked list of marked slot is formed, which is commonly termed as **free list**. Figure 7.7 shows the record of a file with file header pointing to first marked record and so on. New record is placed at the address pointed by file header and header pointer is changed to point next available marked record. In case, no marked record is available, new record is appended at the end of the file.

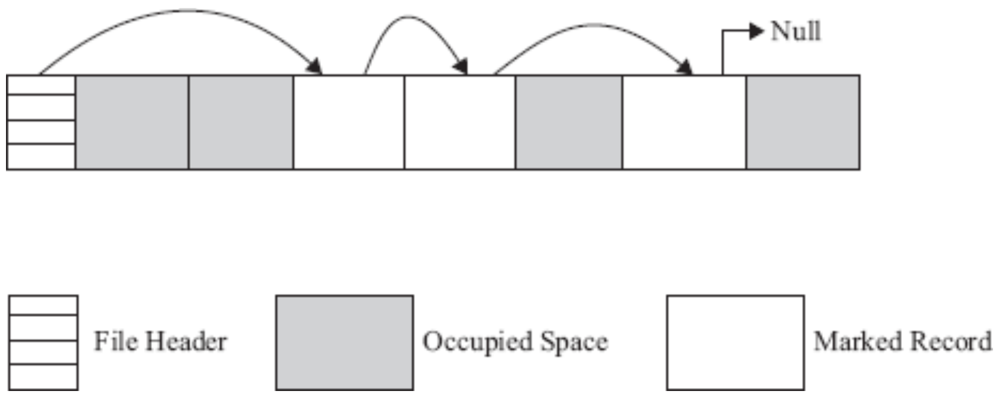


**Fig. 7.7** *Fixed-length records with free list of marked records*

### 7.5.2 Variable-Length Records

Variable-length records may be used to utilize memory more efficiently. In this approach, the exact length of field is not fixed in advance. Thus, to determine the start and end of each field within the record, special separator characters, which do not appear anywhere within the field value, are required (see Figure 7.8). Locating any field within the record requires scan of record until the field is found.

**Fig. 7.8** *Organization of variable-length records with '%' delimiter*

Alternatively, an array of integer offset could be used to indicate the starting address of fields within a record. The `i`$^{th}$ element of this array is the starting address of the `i`$^{th}$ field value relative to the start of the record. An offset to the end of record is also stored in this array, which is used to recognize the end of last field. The organization is shown in Figure 7.9. For *null* value, the pointer to starting and end of field is set same. That is, no space is used to represent a *null* value. This technique is more efficient way to organize the variable-length records. Handling such an offset array is an extra overhead; however, it facilitates direct access to any field of the record.

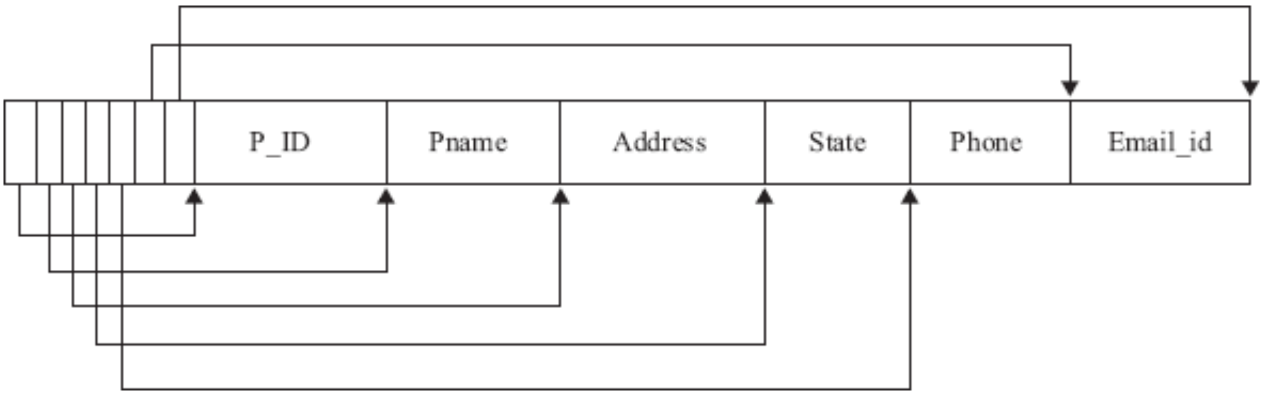


**Fig. 7.9** *Variable-length record organization using an array of field offsets*

Sometimes there may be possibility that the values for a large number of fields are not available or are *null*. In that case, we can store the sequence of pair `<field name, field value>` instead of just field values in each record (see Figure 7.10). In this figure, three separator characters are used—one for separating two fields, second one for separating field value from field name, and third one for separating two records.

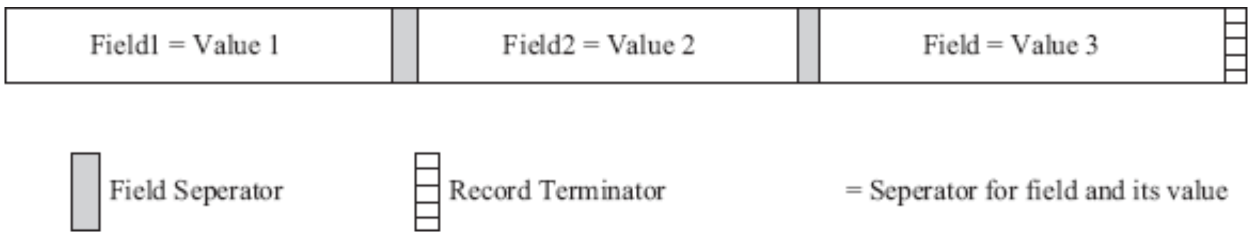


**Fig. 7.10** *Organization for variable-length record*

It is clear from the discussion that the memory is utilized efficiently but processing the variable-length records require complicated program. Moreover, modifying the value of any field might require shifting of all the following fields, since new value may occupy more or less space than the space occupied by the existing value.

### 7.5.3 Mapping Records

After discussing how the database is mapped to files, we will discuss how these file records can be mapped on to disk blocks. There are two ways to organize or place them on to disk blocks. Generally, multiple records are stored in one disk block; however, some files may have large records that cannot fit in one block. Block size is not always multiple of record size, therefore, each disk block might have some unused space. This unused space can be utilized by storing part of a record on one block and the rest on another. In case, consecutive blocks are not allocated to the file, pointer at the end of first block points to the block that contains the remaining part of its last record [see Figure 7.11(a)].

Such type of organization in which records can span more than one disk block is called **spanned organization**. Spanned organization can be used with variable-length records as well as with fixed-length records. On the other hand, in **unspanned organization** the records are not allowed to cross block boundaries, thus, part of each block is wasted [see Figure 7.11(b)]. Unspanned organization is generally used with fixed-length records. This organization ensures starting of records at known position within the block, which makes processing of records simple. It is advantageous to use spanned organization to reduce the lost space in each block if average record is larger than a block.

#### Learn More

In order to simplify the management of free space, most RDBMS impose an upper limit on the record size. However, now-a-days, database often needs to store image, audio, or video objects which are much larger than the size of disk block. In these situations, large objects are stored in separate file(s) and a pointer to them is stored in the record.

*NOTE* Generally records are stored contiguously in the disk block that means next record starts where the previous record ends.

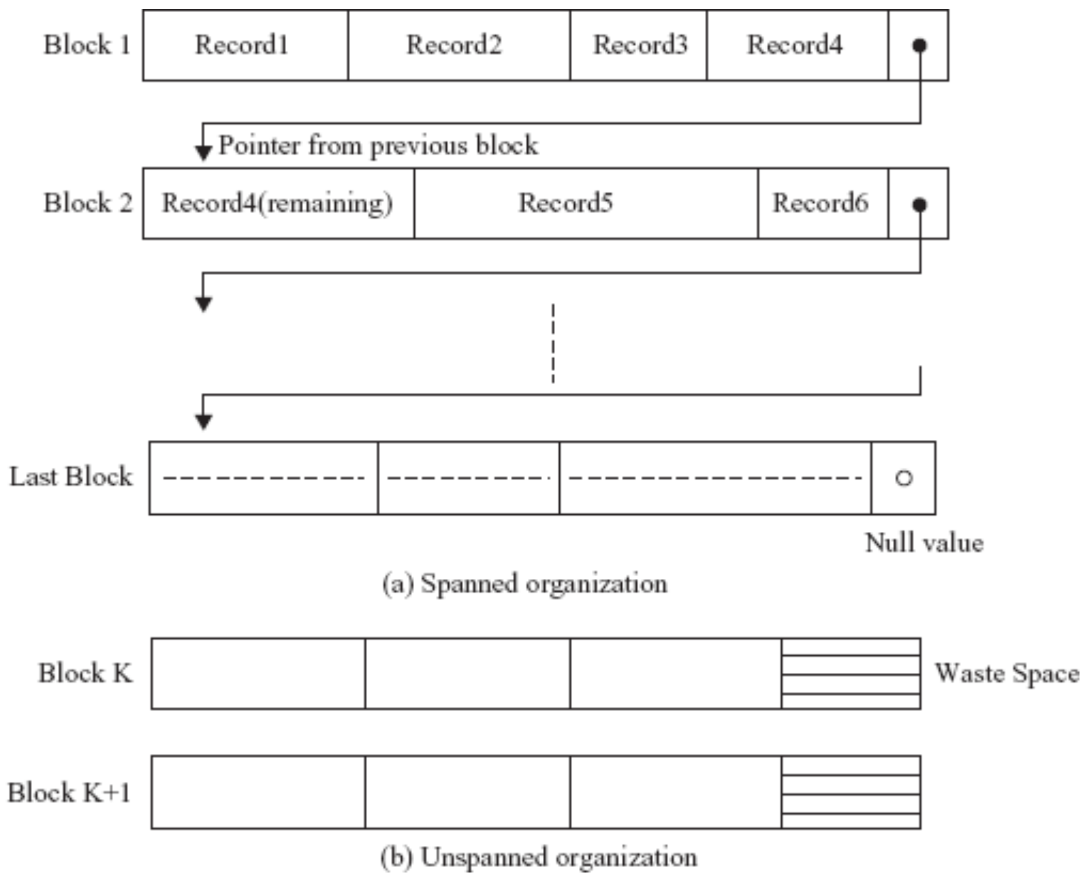


**Fig. 7.11** *Types of record organization*

In case of fixed length record, equal number of records is stored in each block, thus, determining the start and end of record is relatively simple. However, in case of variable-length record, different number of records is stored in each block. Thus, some technique is needed to indicate the start and end of the record within each block. One common technique to implement variable-length records is **slotted page structure**, which is shown in Figure 7.12. Under this technique, some space is reserved for header in the beginning of each block that contains the following information.

- total number of records in the block
- end of free space in the block
- location and size of each record within the block

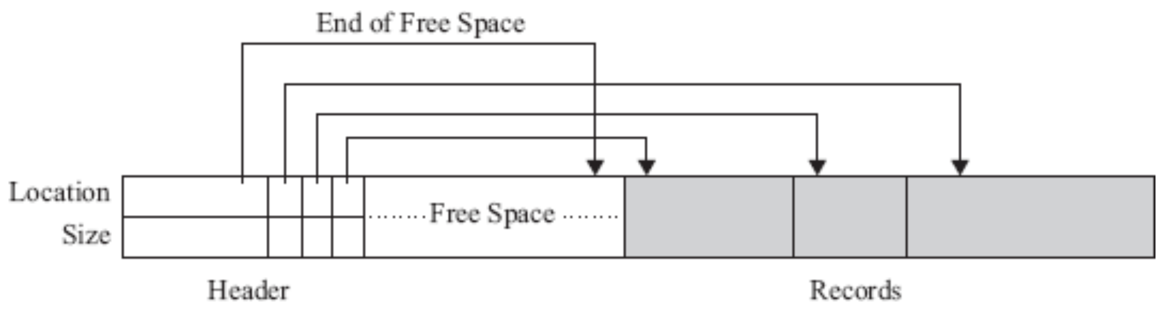


**Fig. 7.12** *Slotted page structure*

In a slotted page structure starting from the end of the block, the actual records are stored contiuously and the free space is contiguous between the final entry in the header array and the first record. The space for new record is allocated at the end of free space and corresponding entry of record location and record size is included in the header. Record is deleted by removing its entry from the header and freeing the space occupied by the record. To make the free space contiguous within the block again, all the records before deleted records are shifted to their right. In addition, header entry pointing to the end of free space is updated as well.

Here, no outside pointers point directly to actual records, rather they must point to the entry in the header, which contains the location and size of each record. This support of indirect pointers to records allows records to be moved within the block to prevent fragmented free space inside a block.

## 7.6 ORGANIZATION OF RECORDS IN FILES

Arrangement of the records in a file plays a significant role in accessing them. Moreover, proper organization of files on disk helps in accessing the file records efficiently. There are various methods (known as **file organization**) of organizing the records in a file while storing a file on disk. Some popular methods are *heap file organization, sequential file organization*, and *hash file organization*.

### 7.6.1 Heap File Organization

It is the simplest file organization technique in which no particular order of record is maintained in the file. Instead, records can be placed anywhere in the file, where there is enough space for that record. Such an organization is generally termed as **heap file** or **pile file**. Heap file is divided into pages of equal size and every record is identified by its unique id or record id.

Operations that can be carried out on a heap file include *creation* and *deletion* of files, *insertion* and *deletion* of record, *searching* a particular record, and *scanning* of all records in the file. Heap file supports efficient insertion of new record; however, scanning, searching, and deletion of records is an expensive process.

Generally, new record is appended at the end of file. The last page of file is copied to the buffer, new record is added and then the page is written back to the disk. The address of last page of file is kept in the file header.

Scanning requires retrieving all the pages of heap file and processing each record on the page one after the other. Let a file has `R` records on each page and every record takes `S` time to process. Further, if the file contains `P` pages and retrieving one page requires `T` time, then total cost to scan the complete file is `P(T+RS)`.

Assuming that exactly one record satisfies the equality selection, it means, the selection is specified on a candidate key whose values are uniformly distributed. On an average, half the file needs to be scanned using linear search technique. If no record satisfies the given condition then, all the records of the file need to be scanned to verify that the searched record is not found.

For deleting a particular record, it needs to be located in the file using its record-id. Record-id helps in identifying the page that contains the record. The page is then transferred in the buffer and after having located the record it is removed from the page and the updated page is written back.

***Implementing Heap Files***

As stated earlier, heap files support various operations. To implement deletion, searching, and scanning operation, there is a need to keep track of all the pages allocated to that file. However, for insertion operation, identification of the pages that contain free space is also required. A simple way to keep this information is to maintain two linked lists of pages—one for those having no free space and another for those having some free space. The system only needs to keep track of first page of the file called header page. One such representation of heap file is shown in Figure 7.13.

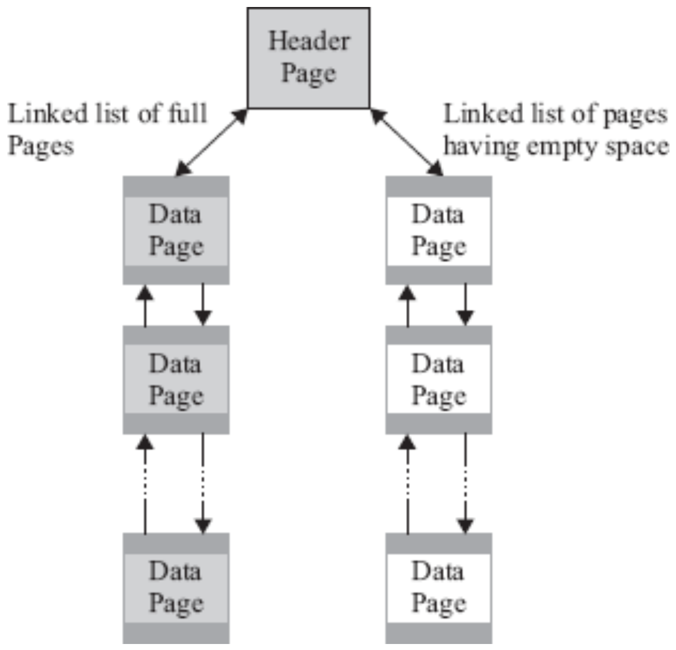


**Fig. 7.13** *Linked list of heap file organization*

When a new record is to be inserted in the file, the page with enough free space is located and the record is inserted in that page. If there is no page with enough space to accommodate that new record, a new page is allocated to the file and record is inserted in the new page.

The main disadvantage of this technique is that we sometimes may require examining several pages for inserting a new record. Moreover, in case of variable length records, each page might always have some free space in it. Thus, all the pages allocated to the file will be on the list of free pages.

This disadvantage of implementing heap file can be addressed by another technique in which a directory of pages is maintained. In this directory, each entry has a pointer to a page in the file and the amount of free space in that page. When directory itself contains several pages, they are linked together to form a linked list. Organization of heap file using directory is shown in Figure 7.14.

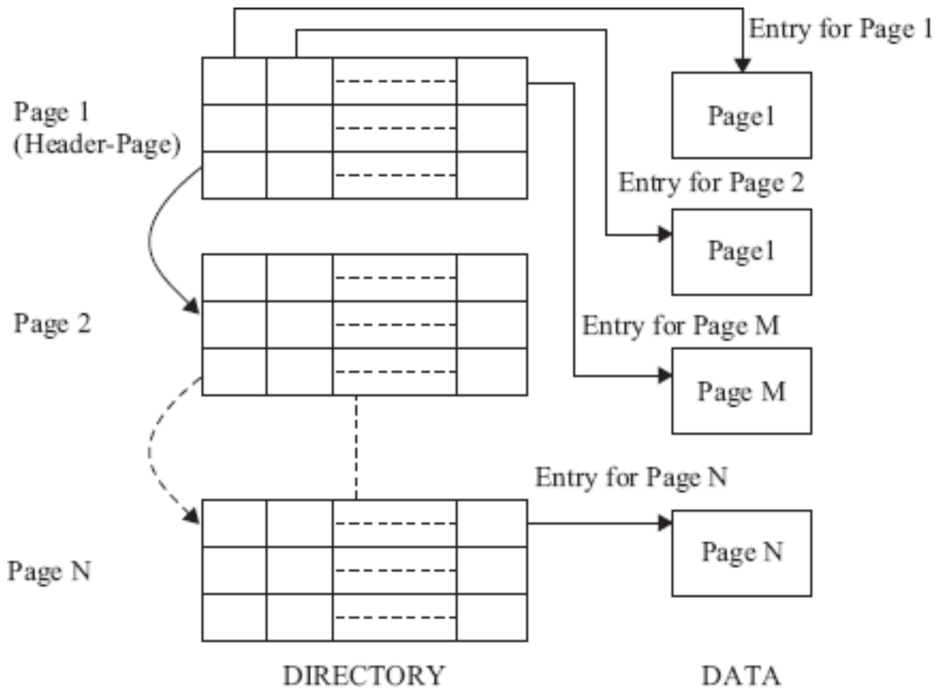


**Fig. 7.14** *Directory-based heap file organization*

### 7.6.2 Sequential File Organization

Often, it is required to process the records of a file in the sorted order based on the value of one of its field. If the records of the file are not physically placed in the required order, it consumes time to fulfill this request. However, if the records of that file are placed in the sorted order based on that field, we would be able to efficiently fulfill this request. A file organization in which records are sorted based on the value of one of its field is called **sequential file organization**, and such a file is called **sequential file**. In a sequential file, the field on which the records are sorted is called **ordered field**. This field may or may not be the key field. In case, the file is ordered on the basis of key, then the field is called the **ordering key**.

Searching of records is more efficient in a sequential file if search condition is specified on the ordering field because then binary search is applicable instead of linear search. Moreover, retrieval of records with the range condition specified on the ordering field is also very efficient. In this operation, all the records starting with the first record satisfying the range selection condition till the first record that does not satisfy the condition are retrieved.

*NOTE* Sequential file does not make any improvement in processing the records in random order.

However, handling deletion and insertion operations are complicated. Deletion of a record leaves a blank space in between the two records. This situation can be handled by using deletion marker in the same way as already discussed in the Section 7.5.1. When a new record is to be inserted in a sequential file, there are two possibilities. First, we can insert the record at its actual position in the file. Obviously, this needs locating the first record that has to come after the new record and making space for the new record. Making space for a record may require shifting a large number of records and this is very costly in terms of disk access. Second, we can insert that record in an overflow area allocated to the file instead of its correct position in the original file. Note that the records in the overflow area are unordered. Periodically, the records in the overflow area are sorted and merged with the records in the original file.

The second approach of insertion makes the insertion process efficient; however, it may affect the search operation. This is because the required record needs to be searched in the overflow area using linear search if it is not found in the original file using binary search.

### 7.6.3 Hash File Organization

In this organization, records are organized using a technique called **hashing**, which enables very fast access to records on the basis of certain search conditions. This organization is called **hash file** or **direct file**. In hashing, a **hash function** `h` is applied to a single field, called **hash field**, to determine the page to which the record belongs. The page is then searched to locate the record. In this way, only one page is accessed to locate a record. Note that the search condition must be an equality condition on the hash field. The hash field is known as **hash key** in case the hash field is the key field of the file.

Hashing is typically implemented as a **hash table** through the use of an array of records. Given the hash field value, the hash function tells us where to look in the array for that record. Suppose that there are `N` locations for records in the array, it means the index of array ranges from 0 to `N–1`. The hash function `h` converts the hash field value between 0 and `N–1`. Note that non-integer hash field value can be converted into integer before applying hash function. For example, the numeric (ASCII) code associated with characters can be used in converting character values into integers. Some common hash functions are discussed here.

**Things to Remember**

Locating a record for equality search conditions is very effective using hashing technique. However, with range conditions, it is almost as worse as scanning all the records of the file.

- **Cut key hashing:** This hash function 'cuts' some digits from the hash field value (or simply key) and uses it as hash address. For example, if there are 100 locations in the array, a 2-digit address, say 37, may be obtained from the key 1324 **37** by picking the highlighted digits. It is not a good algorithm because most of the key is ignored and only a part of the key is used to compute the address.
- **Folded key:** This hash function involves applying some arithmetic functions, such as *addition/ subtraction* or some logical functions, such as *and/or* to the key value. The result obtained is used as the record address. For example, the key is broken down into a group of 2-digit numbers from the left most digits and they are added like 13 + 24 + 37 = 74. The sum is then used as record address. It is better than cut key hashing as the entire key is used but not good enough as records are not distributed evenly among pages.
- **Division-remainder hashing:** This hash function is commonly used

and in this the value of hash field is divided by `N` and remainder is used as record address. Record `address = (Key value) mod (N).`

For example, `(132437) mod (101) = 26`

This technique works very well provided that `N` is either a prime number or does not have a small divisor.

The main problem associated with most hashing functions is that they do not yield distinct addresses for distinct hash field values, because the number of possible values a hash field can take is much larger than the number of available addresses to store records. Thus, sometimes a problem, called **collision**, occurs when hash field value of a record to be inserted hashes to an address which is already being occupied by another record. It should be resolved by finding some other location to place the new record. This process of finding another location is called **collision resolution**. Some popular methods for collision resolution are given here.

- **Open addressing:** In this method, the program keeps checking the subsequent locations starting from the occupied location specified by hash function until an unused or empty location is found.
- **Multiple hashing:** In this method, the program can use another hash function if the first hash function results in a collision. If another collision occurs, the program may apply another hash function and so on, or it can use open addressing if necessary. This technique of applying multiple hash function is called **double hashing** or **multiple hashing**.
- **Chained overflow:** In this method, all the records whose addresses are already occupied are stored in an overflow space. In addition, a pointer field is associated with each record location. A collision is resolved by placing the new record in overflow space and the pointer field of occupied hash address location is set to that location in overflow space. In this way, a linked list of records, which hash to same address, is maintained, as shown in Figure 7.15.

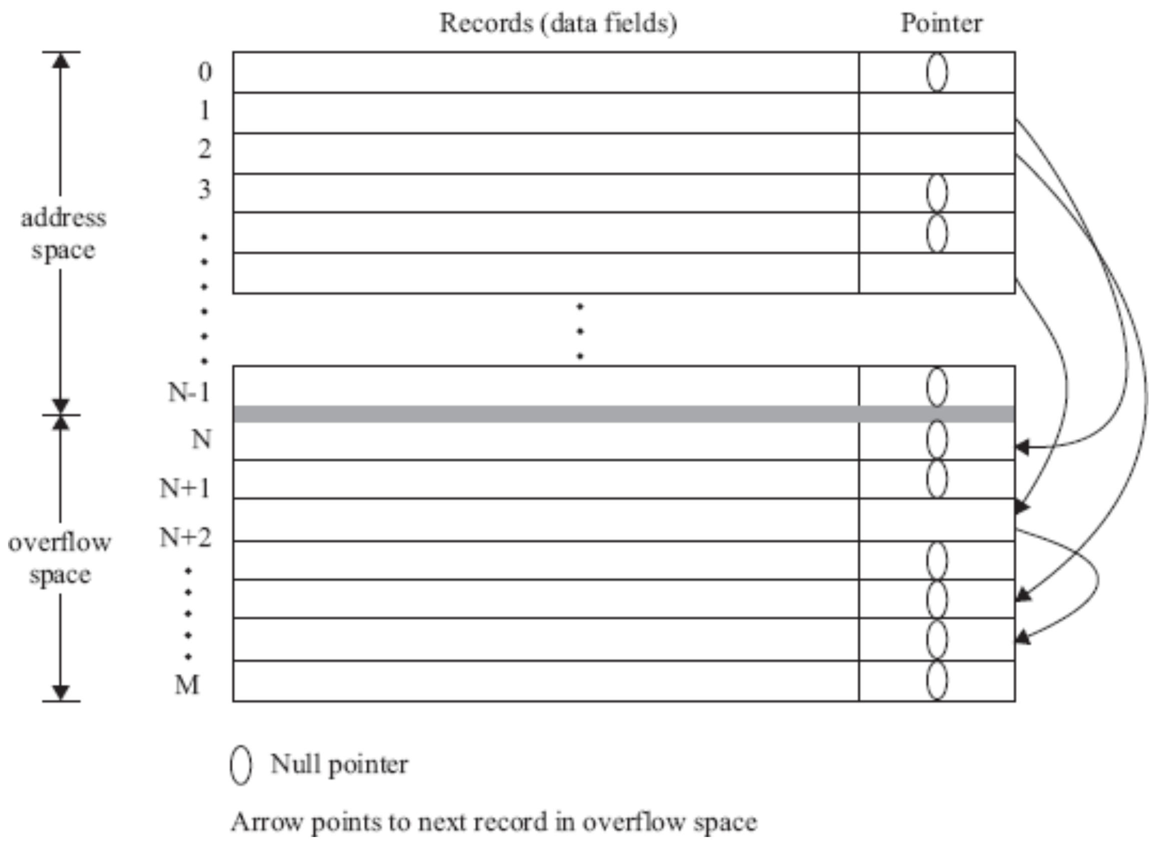


**Fig. 7.15** *Collision resolution using chained overflow*

All these methods work well; however, each of them requires its own algorithms for insertion, retrieval, and deletion of records. All the algorithms for chained overflow are simple, whereas open addressing requires tricky algorithm for deletion of records.

A good hash function must distribute records uniformly over the available addresses so as to minimize collisions. In addition, it should always return same value for same input. Generally, hashing works best when division remainder hash function is used in which `N` is chosen as a prime number, as it distributes the records better over the available addresses.

Further, the collision problem is less severe by using the concept of **buckets**, which represents the storage unit that can store one or more records. There are two types of hashing schemes namely, *static hashing* and *dynamic hashing*, depending on the number of buckets allocated to a file.

### ***Static Hashing***

The hashing scheme in which a fixed number of buckets, say `N`, are allocated to a file to store records is called **static hashing**. The pages of a file can be viewed as a collection of buckets, with one **primary** page and additional **overflow** pages. The file consists of `0` to `N–1` buckets, with one primary page per bucket initially. Further, let `V` denote the set of all values hash field can take then the hash function `h` maps `V` to `N`.

To insert a record, hash function is applied on the hash field to identify the bucket to which the record belongs and then it is placed there. If the bucket is already full, a new overflow page is allocated and the record is placed in the new overflow page and then the page is added to the **overflow chain** of bucket.

To search a record, the hash function transforms the hash field value to the bucket to which the required record belongs. Hash field value of all the records in the bucket is then verified to search the required record. Note that to speed up the searching process within the bucket, all the records are maintained in a sorted order by hash field value.

To delete a record, hash function is applied to identify the bucket to which it belongs. The bucket is then searched to locate the corresponding record and after locating the record, it is removed from the bucket. Note that the overflow page is removed from the overflow chain of the bucket if it is the last record in the overflow page.

Figure 7.16 illustrates the static hashing which represents the records of `BOOK` relation with `Price` as hash field. In this example, the hash function `h` takes the remainder of `Price` after dividing it by three to determine the bucket for a record. Function `h` is defined as

> `h(Price) = 0`, record for bucket 1
>
> `h(Price) = 1`, record for bucket 2
>
> `h(Price) = 2`, record for bucket 3

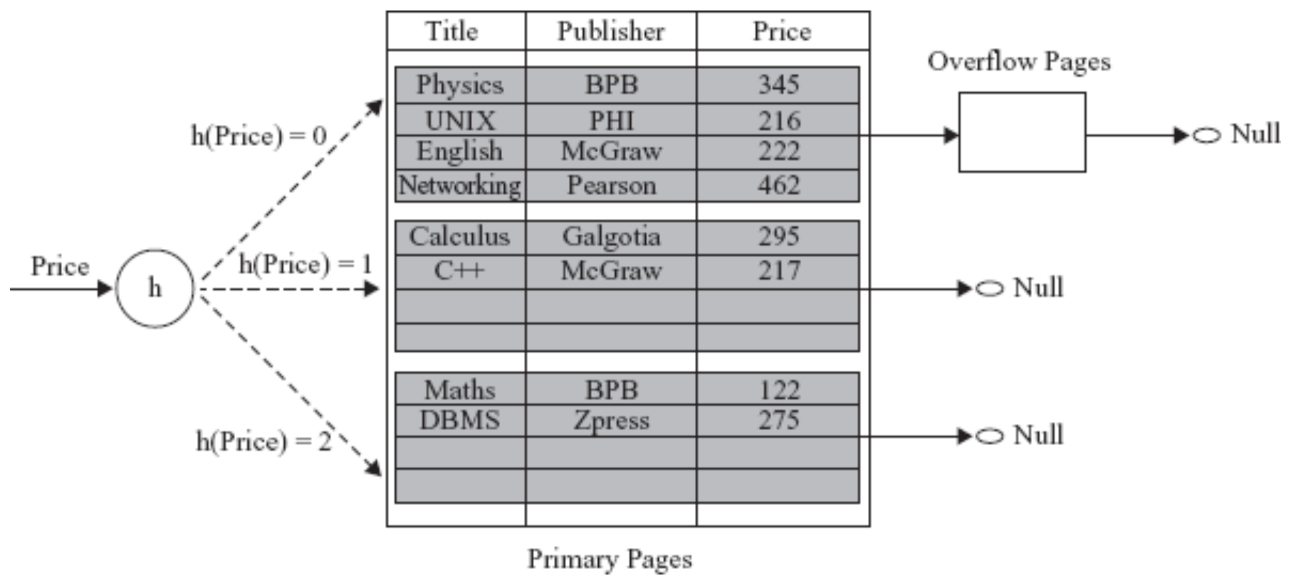


**Fig. 7.16** *Static hashing with* `Price` *as hash field*

In static hashing, since the number of buckets allocated to a file are fixed when the file is created, the primary pages can be stored on consecutive disk pages. Hence, searching a record requires just one disk I/O, and other operations, like insertion and deletion, require two I/Os (read and write the page). However, as the file grows, long overflow chains are developed which degrade the performance of the file as searching a bucket requires searching all the pages in its overflow chain.

It is undesirable to use static hashing with files that grow or shrink a lot dynamically, since number of buckets is fixed in advance. It is a major drawback for dynamic files. Suppose a file grows substantially more than the allocated space, a long overflow chain is developed which results in poor performance. Similarly, if a file shrinks greatly, a lot of space is left unused. In either case, the number of buckets allocated to the file needs to be changed dynamically and new hash function based on new value of `N` should be used for distribution of records. However, such reorganization consumes a lot of time for large files. Another alternative is to use dynamic hashing, which allows number of buckets to vary dynamically with only minor (internal) reorganization.

### *Dynamic Hashing*

As discussed earlier, the number of available addresses for records is fixed in advance in static hashing, which makes it unsuitable for the files that grow or shrink dynamically. Thus, **dynamic hashing** technique is needed which allocates new buckets dynamically as needed. In addition, it allows the hash function to be modified dynamically to accommodate the growth or shrinkage of database files. Dynamic hashing is of two forms, namely, *extendible hashing* and *linear hashing*. In our discussion of dynamic hashing, we consider the result of hash function as binary representation of hash field value.

*Extendible Hashing* Extendible hashing uses a **directory** of pointers to buckets. A directory is just an array of $2^d$ size, where `d` (called **global depth**) is the number of bits of hash value used to locate the directory element. Each element of the directory is a pointer to a bucket in which the corresponding records are stored.

In this technique, the hash function is applied on the hash field value and the last `d` bits of hash value are used to locate the directory element. The pointer in this array position points to the bucket to which the corresponding record belongs. To understand this technique, consider a directory consisting of an array of $2^2$ size (see Figure 7.17), which means an array of size 4. Here `d` is 2, thus, last 2 bits of hash value are used to locate a directory element. Further, assume that each bucket can hold three records. The search for a key, say 21, proceeds as follows. Last 2 bits of hash value 21 (binary 10101) hashes into the directory element 01, which points to bucket 2.

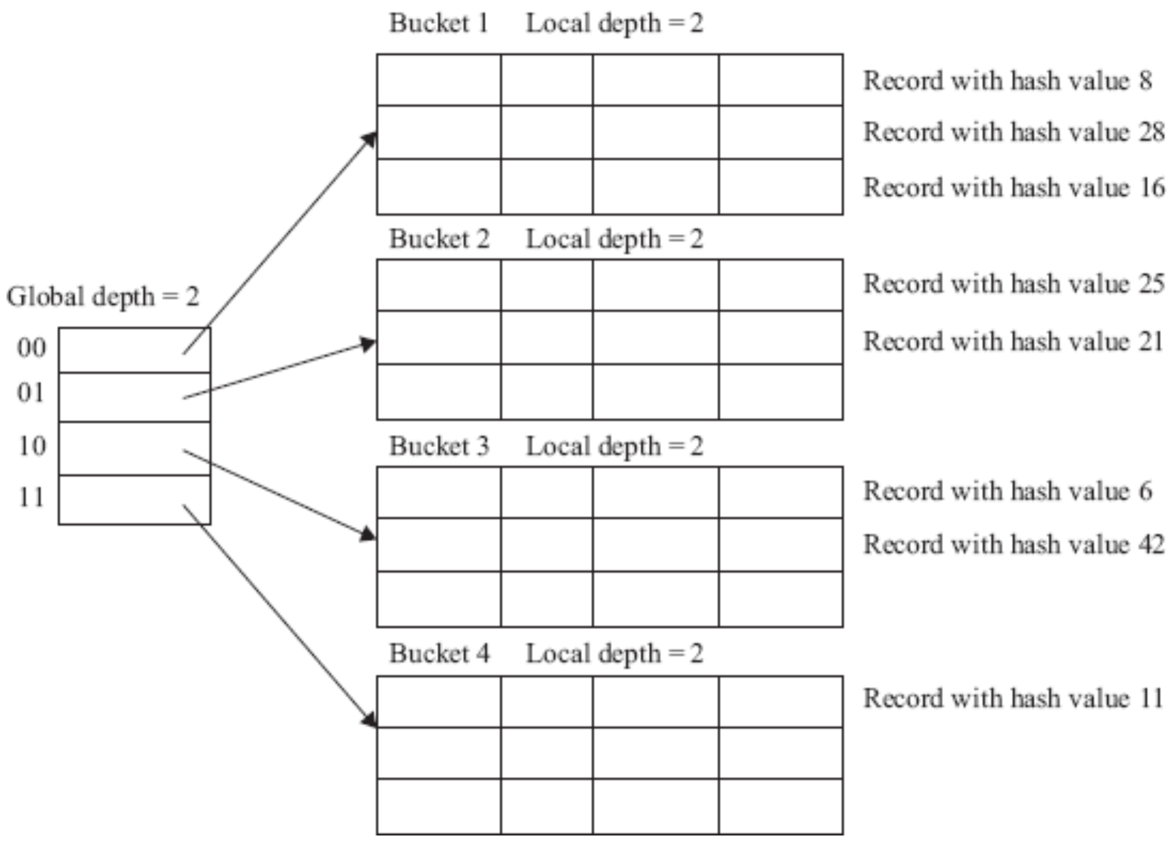


**Fig. 7.17** *Extendible hashed file*

Consider the insertion of a new record with hash field value 9 (binary 1001). It hashes to the directory element 01, which further points to the bucket 2. Since bucket 2 has space for a new record, it is inserted there. Next, consider the insertion of a record with hash field value 24 (binary 11000). It hashes to the directory element 00 which points to bucket 1, which is already full. In this situation, extendible hashing splits this full bucket by allocating a new bucket and redistributes the records across the old bucket and its split image. To redistribute the records among these two buckets, the last three bits of hash value are considered. The last two bits (00 in this case) indicate the directory element and the third bit differentiate between these two buckets. Splitting of bucket with redistributing of records is shown in Figure 7.18. Now, the contents of bucket 1 and bucket 1a are identified by 3 bits of hash value, whereas the contents of all other buckets can be identified by 2 bits. To identify the number of bits on which the bucket contents are based, a **local depth** is maintained with each bucket, which specifies the number of bits required to identify its contents.

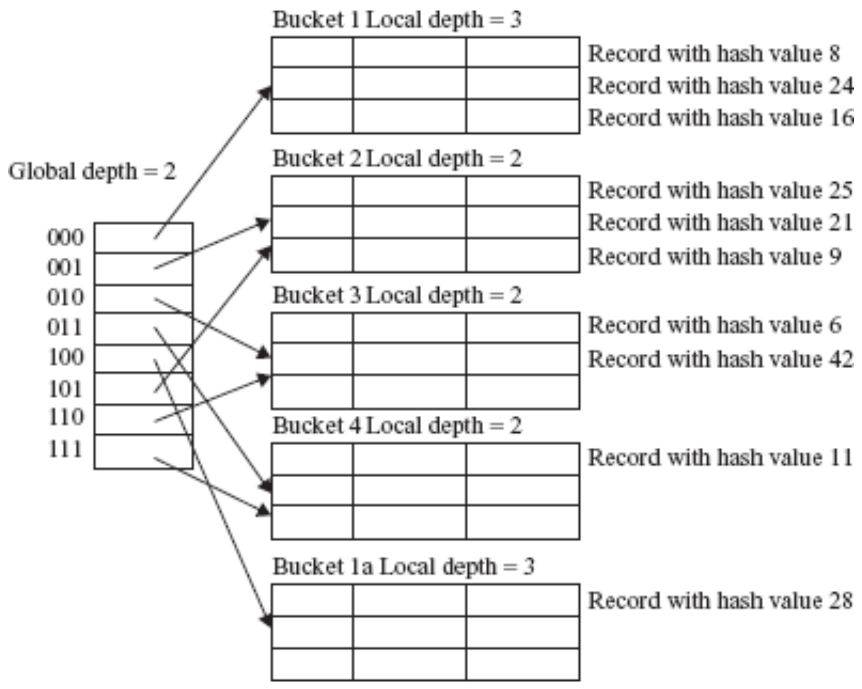


**Fig. 7.18** *Extendible hashing with splitting bucket*

Note that if the local depth of overflowed bucket is equal to the global depth there is a need to double the number of entries in the directory and global depth is incremented by one. However, if local depth of overflowed bucket is less than the global depth there is no need to double the directory entries. For example, consider the insertion of a record with hash value 5 (binary 101). It hashes to the directory element 01 which points to bucket 2, which is already full. This situation is handled by splitting the bucket 2 and using the directory element 001 and 101 to point to the bucket 2 and its split image, respectively, (see Figure 7.19).

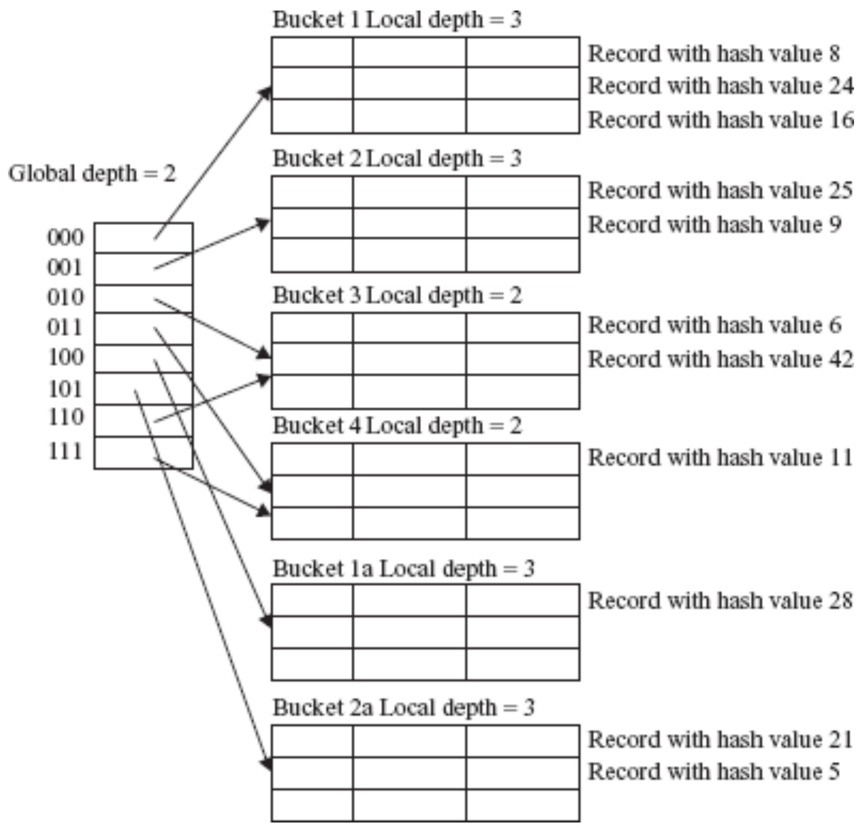


**Fig. 7.19** *After inserting record with hash value 5*

To delete a record, first the record is located and then removed from the bucket. In case, the deletion leaves the bucket empty the bucket can be merged with its split bucket image. The local depth is decreased if the buckets are merged. After merging, if the local depth of all the buckets becomes less than the global depth, the number of entries in the directory can be halved and global depth is be reduced by one.

The main advantage of extendible hashing over static hashing is that the performance of the file does not degrade as the file grows dynamically. In addition, there is no space allocated in advance for future growth of the file. However, buckets can be allocated dynamically as needed. The size of directory is likely to be much smaller than the file itself, since each directory element is just a page-id. Thus, the space overhead for the directory table is negligible. The directory can be extended up to $2^n$, where $n$ is the number of bits in the hash value. Another advantage of extendible hashing is that bucket splitting requires minor reorganization in most cases. The reorganization is expensive when directory needs to be doubled or halved. One disadvantage of extendible hashing is that the directory must be accessed and searched before accessing the record in the bucket. Thus, most record retrievals require two block accesses, one for the directory and the other for the bucket.

*Linear Hashing* **Linear hashing** also allows a hash file to grow or shrink dynamically by allocating new buckets. In addition, there is no need to maintain a directory of pointers to buckets. Like static hashing, collision can be handled by maintaining overflow chains of pages; however, linear hashing solves the problem of long overflow chains. In linear hashing, overflow of any bucket in the file leads to a bucket split. Note that the bucket to be split is not necessarily the same as the overflow bucket, instead buckets are chosen to be split in the linear order 0, 1, 2, ....

To understand the linear hashing scheme, consider a file with `T` buckets, initially numbered 0 through `T–1`. Suppose that the file uses a `mod` hash function $h_i$, denoted by `h(V) = V mod T`. Linear hashing uses a value `j` to determine the bucket to be split. Initially `j` is set to 0 and is incremented by 1 each time a split occurs. Thus, whenever an insert triggers an overflow record in any bucket, the $j^{th}$ bucket in the file is split into two buckets, namely, the bucket `j` and a new bucket `T + j` and `j` is incremented by 1. In addition, all the records of original bucket `j` are redistributed among the bucket `j` and the bucket `T + j` using a new hash function $h_{i+1}$, denoted by $h_{i+1}$ `(V) = V mod 2T`. Thus, when all the original `T` buckets have been split, all buckets use the hash function $h_{i+1}$.

A key property of hash function $h_{i+1}$ is that any record that is hashed to bucket `j` by $h_i$ will hash either to bucket `j` or bucket `T + j`. The range of hash function $h_{i+1}$ is twice as that of $h_i$, that is, if $h_i$ hashes a record into one of `T` buckets, $h_{i+1}$ hash that record in one of `2T` buckets.

In order to search a record with hash key value `V`, first the hash function $h_i$ is applied to `V` and if $h_i$ `(V) < j`, it means that the hashed bucket is already split. Then, the hash function $h_{i+1}$ is applied to `V` to determine which of the two split buckets contain the record.

Since `j` is incremented with each split, so, when `j = T`, it indicates that all the original buckets have been split. At this point, `j` is reset to 0 and any overflow leads to the use of new hash function $h_{i+2}$, denoted by $h_{i+2}(V) = V \bmod 4T$. In general, linear hashing scheme uses a family of hash functions $h_{i+k}(V) = V \bmod (2^k T)$, where $k$ = 0, 1, 2, .... Each time when all the original buckets 0 through $(2^k T) - 1$ have been split, $k$ is incremented by 1 and new hashing function $h_{i+k}$ is needed.

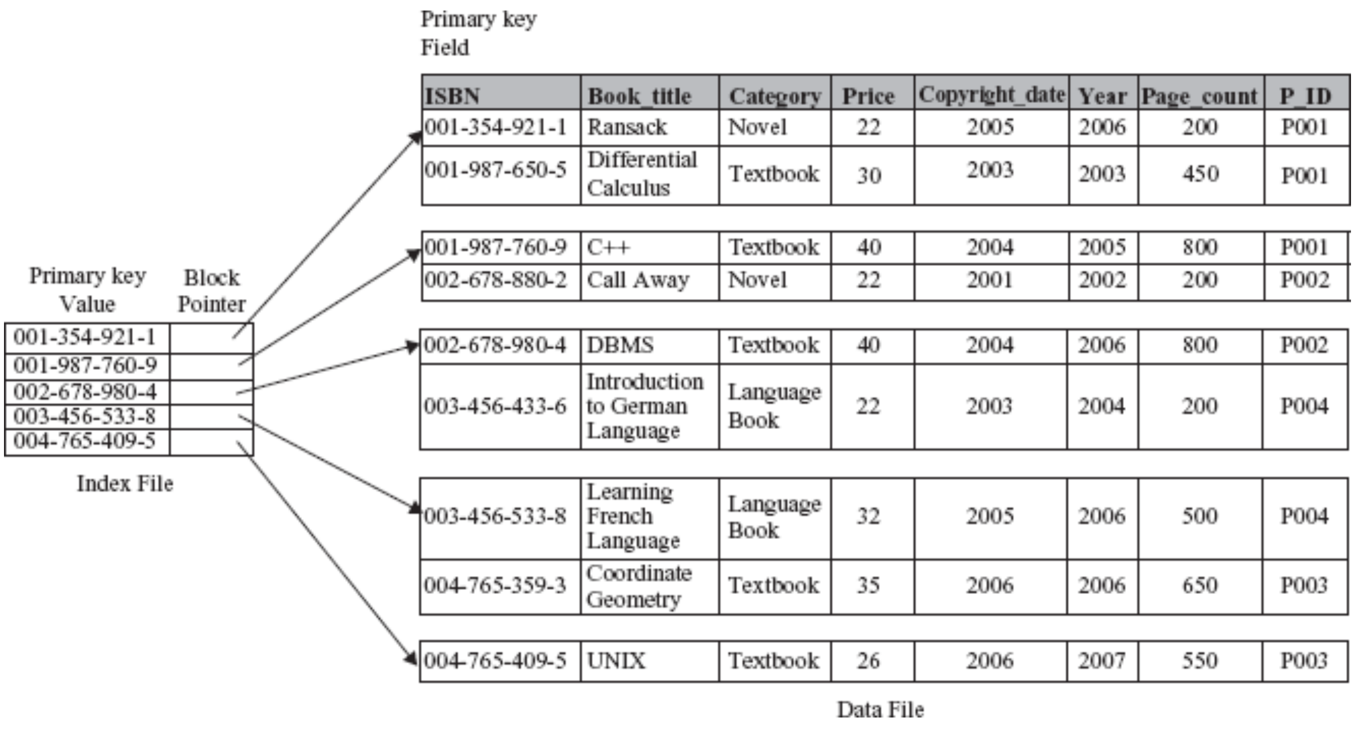
## 7.7 INDEXING

Once the records of a file are placed on the disk using some file organization method, the main issue is to provide quick response to different queries. For this, additional structure called **index** on the file can be created. Creating an index on a file does not affect the physical placement of the records but still provides efficient access to the file records. Index can be created on any field of the file, and that field is called **indexing field** or **indexing attribute**. Further, more than one index can be created on a file.

Accessing records of a file using index requires accessing the index, which helps us to locate the record efficiently. This is because the values in the index are stored in the sorted order and the size of the index file is small as compared to the original file. Thus, applying binary search on index is efficient as compared to applying binary search on the original file. Different types of indexes are possible, which we discuss in this section.

### 7.7.1 Single-Level Indexes

Indexes can be created on the field based on which the file is ordered as well as on the field based on which the file is not ordered. Consider an index created on the ordered attribute ISBN of the BOOK relation (see Figure 7.20). This index is called **primary index**, since it is created on the primary key of the relation.



| Primary key Value | Block Pointer |
|---|---|
| 001-354-921-1 | |
| 001-987-760-9 | |
| 002-678-980-4 | |
| 003-456-533-8 | |
| 004-765-409-5 | |

| ISBN | Book_title | Category | Price | Copyright_date | Year | Page_count | P_ID |
|---|---|---|---|---|---|---|---|
| 001-354-921-1 | Ransack | Novel | 22 | 2005 | 2006 | 200 | P001 |
| 001-987-650-5 | Differential Calculus | Textbook | 30 | 2003 | 2003 | 450 | P001 |
| 001-987-760-9 | C++ | Textbook | 40 | 2004 | 2005 | 800 | P001 |
| 002-678-880-2 | Call Away | Novel | 22 | 2001 | 2002 | 200 | P002 |
| 002-678-980-4 | DBMS | Textbook | 40 | 2004 | 2006 | 800 | P002 |
| 003-456-433-6 | Introduction to German Language | Language Book | 22 | 2003 | 2004 | 200 | P004 |
| 003-456-533-8 | Learning French Language | Language Book | 32 | 2005 | 2006 | 500 | P004 |
| 004-765-359-3 | Coordinate Geometry | Textbook | 35 | 2006 | 2006 | 650 | P003 |
| 004-765-409-5 | UNIX | Textbook | 26 | 2006 | 2007 | 550 | P003 |

**Fig. 7.20** *Index on the attribute* `ISBN` *of* `BOOK` *relation*

This index file contains two attributes—first attribute stores the value of the attribute on which the index is created and second attribute contains a pointer to the record in the original file. Note that it contains an entry for the first record in each disk block instead of every value of the indexing attribute. Such types of indexes which do not include an entry for each value of the indexing attribute are called **sparse indexes** (or **non-dense indexes**). On the other hand, the indexes which include an entry for each value of the indexing attribute are called **dense indexes**.

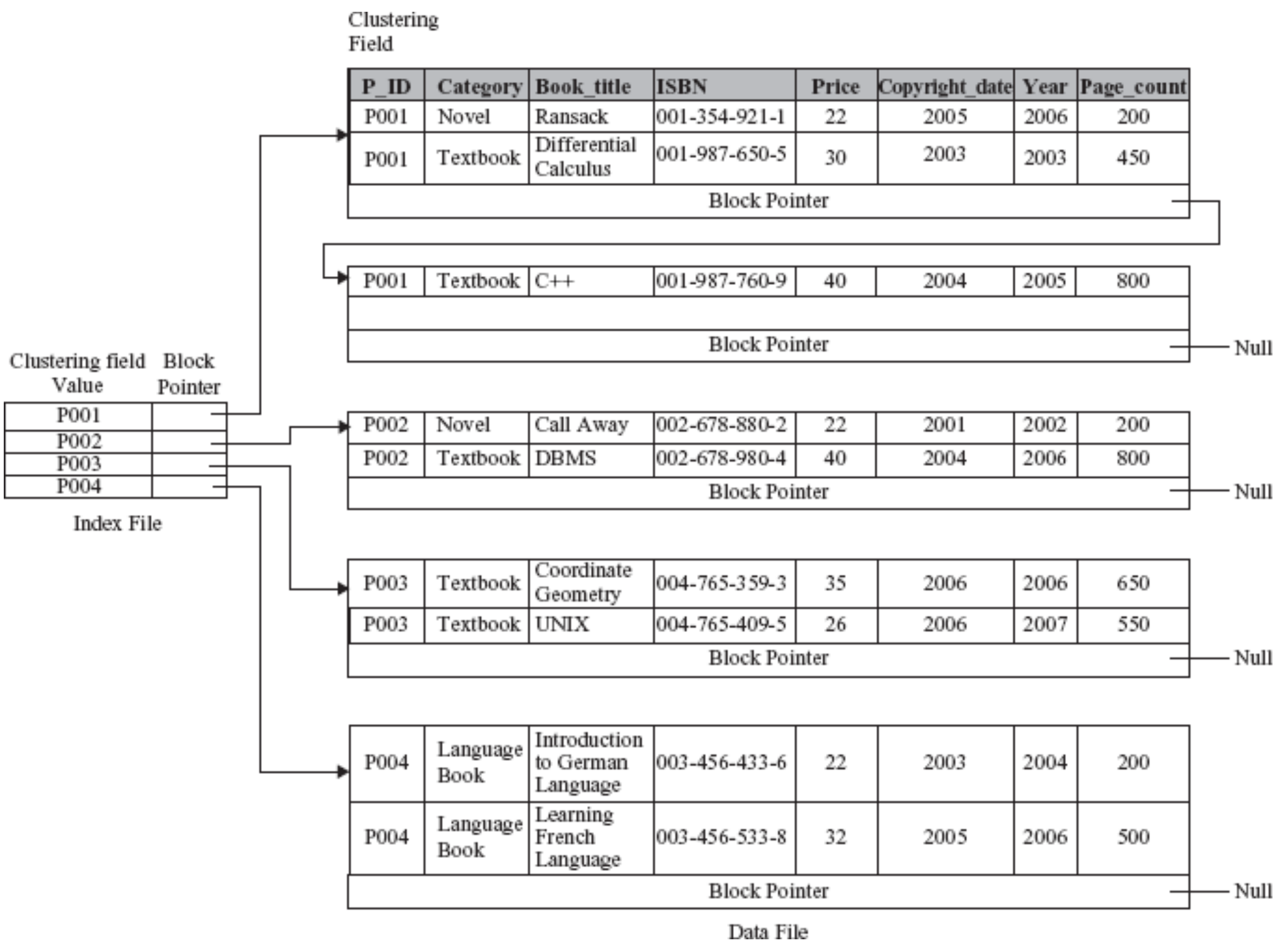
Given the search-key value for accessing any record using primary index, we locate the largest value in the index file which is less than or equal to the search-key value. The pointer corresponding to this value directs us to the first record of the disk block in which the required record is placed (if available). If the first record is not the required record, we sequentially scan the entire disk block in order to find that record. Insertion and deletion operation on the ordered file is handled in the same way as discussed in the sequential file. Here; however, it may also require modifying the pointers in several entries of the index file.

It is not always the case that index is created on the primary key attribute. If an index is created on the ordered non-key attribute, then it is called

**clustering index** and that attribute is called **clustering attribute**. Figure 7.21 shows a clustering index created on the non-key attribute `P_ID` of the `BOOK` relation.

**Learn More**

Though dense index provides faster access to the file records than sparse index, having a sparse index with one entry per block is advantageous in terms of disk space and maintenance overhead while update operations. Moreover, once the disk block is identified and its records are brought into the memory, scanning them is almost negligible.



| Clustering field Value | Block Pointer |
|---|---|
| P001 | |
| P002 | |
| P003 | |
| P004 | |

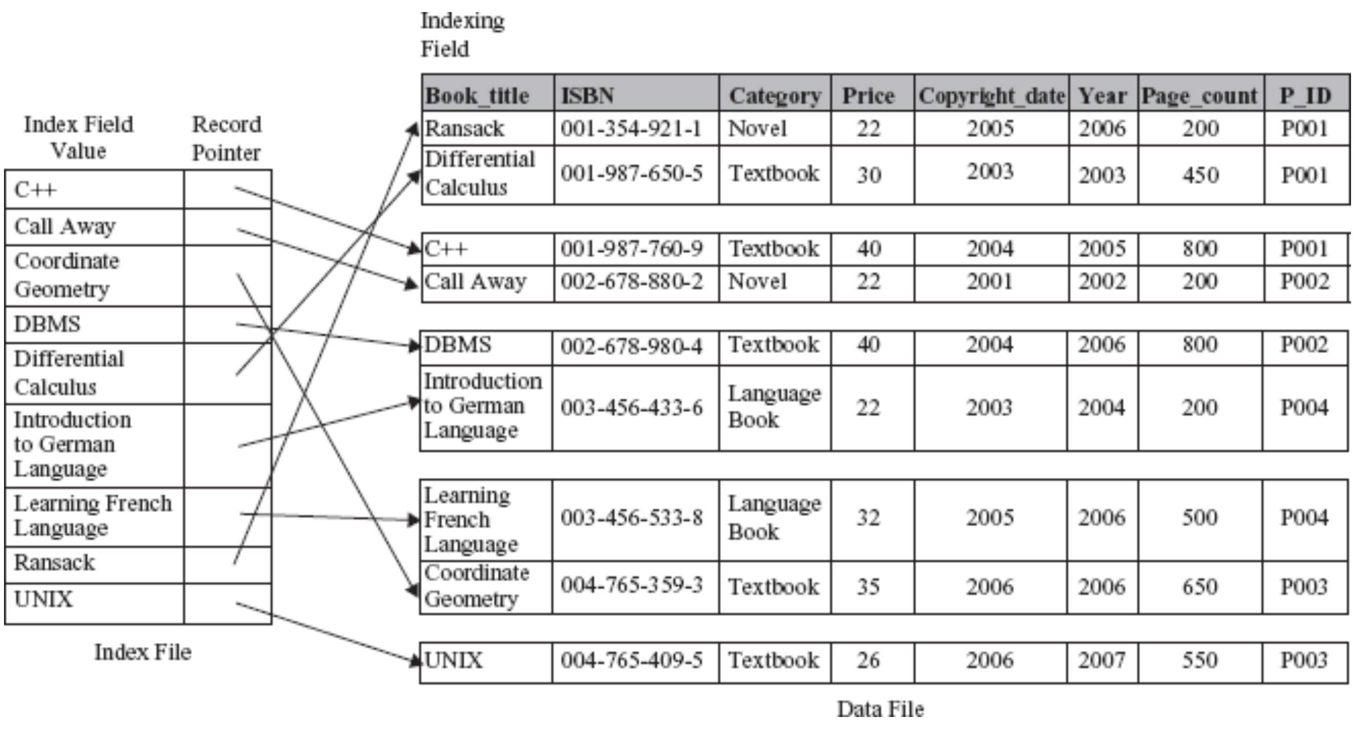
| P_ID | Category | Book_title | ISBN | Price | Copyright_date | Year | Page_count |
|---|---|---|---|---|---|---|---|
| P001 | Novel | Ransack | 001-354-921-1 | 22 | 2005 | 2006 | 200 |
| P001 | Textbook | Differential Calculus | 001-987-650-5 | 30 | 2003 | 2003 | 450 |
| Block Pointer | | | | | | | |
| P001 | Textbook | C++ | 001-987-760-9 | 40 | 2004 | 2005 | 800 |
| | | | | | | | |
| Block Pointer | | | | | | | |
| P002 | Novel | Call Away | 002-678-880-2 | 22 | 2001 | 2002 | 200 |
| P002 | Textbook | DBMS | 002-678-980-4 | 40 | 2004 | 2006 | 800 |
| Block Pointer | | | | | | | |
| P003 | Textbook | Coordinate Geometry | 004-765-359-3 | 35 | 2006 | 2006 | 650 |
| P003 | Textbook | UNIX | 004-765-409-5 | 26 | 2006 | 2007 | 550 |
| Block Pointer | | | | | | | |
| P004 | Language Book | Introduction to German Language | 003-456-433-6 | 22 | 2003 | 2004 | 200 |
| P004 | Language Book | Learning French Language | 003-456-533-8 | 32 | 2005 | 2006 | 500 |
| Block Pointer | | | | | | | |

**Fig. 7.21** *Clustering index on `P_ID` field of `BOOK` relation*

This index contains an entry for each distinct value of the clustering attribute and the corresponding pointer points to the first record with that clustering attribute value. Other records with the same clustering attribute value are stored sequentially after that record, since the file is ordered on that attribute.

Inserting a new record in the file is easy if space is available in the block in which it has to be placed. Otherwise, a new block is allocated to the file and the new record is placed in that block. In addition, the pointer of the block to which the record actually belongs is made to point to the new block.

So far, we have discussed the case in which the index is created on the ordered attribute of the file. Indexes can also be created on the non-ordered attribute of the file. Such indexes are called **secondary indexes**. The indexing attribute may be a candidate key or a non-key attribute with duplicate values. In either case, secondary index must contain an entry for each value of the indexing attribute. It means the secondary index must be a dense index. This is because the file is not ordered on the indexing attribute and if some of the values are stored in the index, it is not possible to find a record whose entry does not exist in the index. Figure 7.22 shows a secondary index created on the non-ordered attribute `Book_title` of `Book` relation.



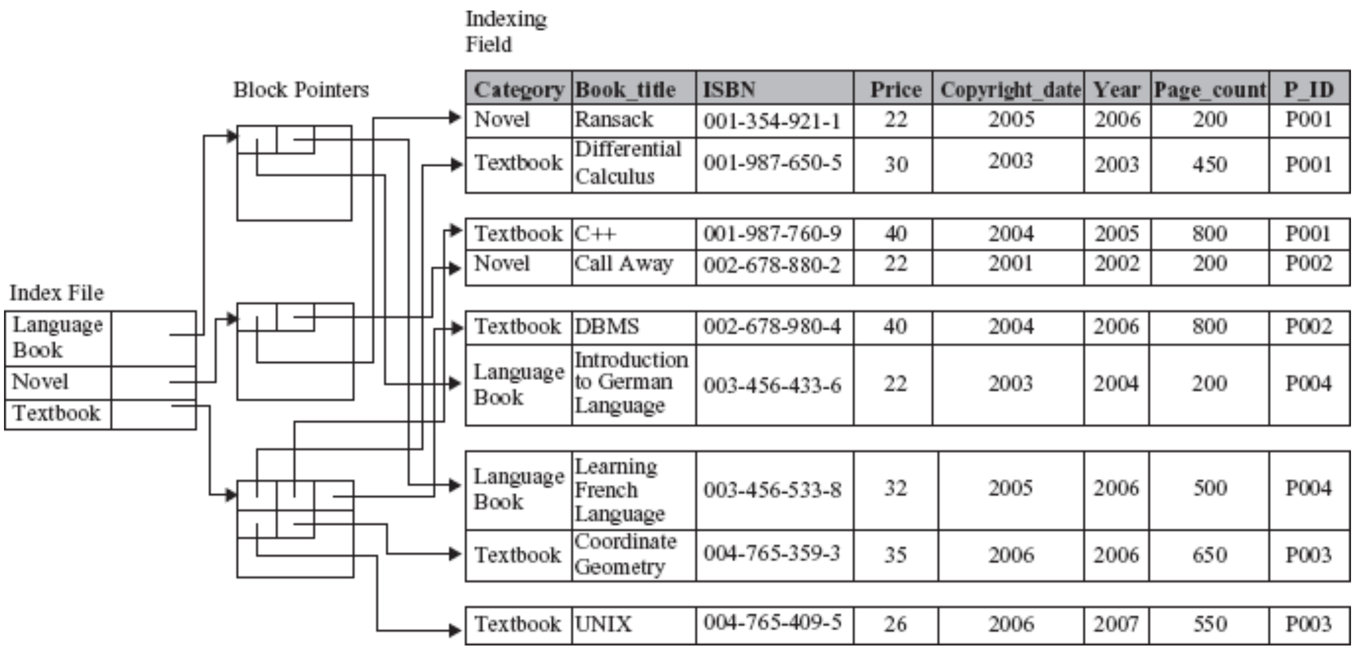
| Index Field Value | Record Pointer |
|---|---|
| C++ | |
| Call Away | |
| Coordinate Geometry | |
| DBMS | |
| Differential Calculus | |
| Introduction to German Language | |
| Learning French Language | |
| Ransack | |
| UNIX | |

| Book_title | ISBN | Category | Price | Copyright_date | Year | Page_count | P_ID |
|---|---|---|---|---|---|---|---|
| Ransack | 001-354-921-1 | Novel | 22 | 2005 | 2006 | 200 | P001 |
| Differential Calculus | 001-987-650-5 | Textbook | 30 | 2003 | 2003 | 450 | P001 |
| C++ | 001-987-760-9 | Textbook | 40 | 2004 | 2005 | 800 | P001 |
| Call Away | 002-678-880-2 | Novel | 22 | 2001 | 2002 | 200 | P002 |
| DBMS | 002-678-980-4 | Textbook | 40 | 2004 | 2006 | 800 | P002 |
| Introduction to German Language | 003-456-433-6 | Language Book | 22 | 2003 | 2004 | 200 | P004 |
| Learning French Language | 003-456-533-8 | Language Book | 32 | 2005 | 2006 | 500 | P004 |
| Coordinate Geometry | 004-765-359-3 | Textbook | 35 | 2006 | 2006 | 650 | P003 |
| UNIX | 004-765-409-5 | Textbook | 26 | 2006 | 2007 | 550 | P003 |

**Fig. 7.22** *Secondary index with record pointers on a non-ordering key field of* BOOK *relation*

This index contains an entry for each value of the indexing attribute and the corresponding pointer points to the record in the file. Note, that the index file itself is ordered on the indexing attribute. Inserting and deleting a record is straightforward with a little modification in the index file.

If the indexing attribute of the secondary index is a non-key attribute with duplicate values, then we may use an extra level of indirection to handle multiple pointers. In this case, the pointer in the index points to a bucket of pointers instead of the actual records in the file. Each pointer in the bucket either points to the records in the file or to the disk block that contains the record. This structure of secondary index is illustrated in Figure 7.23.



| Category | Book_title | ISBN | Price | Copyright_date | Year | Page_count | P_ID |
|---|---|---|---|---|---|---|---|
| Novel | Ransack | 001-354-921-1 | 22 | 2005 | 2006 | 200 | P001 |
| Textbook | Differential Calculus | 001-987-650-5 | 30 | 2003 | 2003 | 450 | P001 |
| Textbook | C++ | 001-987-760-9 | 40 | 2004 | 2005 | 800 | P001 |
| Novel | Call Away | 002-678-880-2 | 22 | 2001 | 2002 | 200 | P002 |
| Textbook | DBMS | 002-678-980-4 | 40 | 2004 | 2006 | 800 | P002 |
| Language Book | Introduction to German Language | 003-456-433-6 | 22 | 2003 | 2004 | 200 | P004 |
| Language Book | Learning French Language | 003-456-533-8 | 32 | 2005 | 2006 | 500 | P004 |
| Textbook | Coordinate Geometry | 004-765-359-3 | 35 | 2006 | 2006 | 650 | P003 |
| Textbook | UNIX | 004-765-409-5 | 26 | 2006 | 2007 | 550 | P003 |

**Fig. 7.23** *Secondary index on a non-key field using an extra level of indirection*

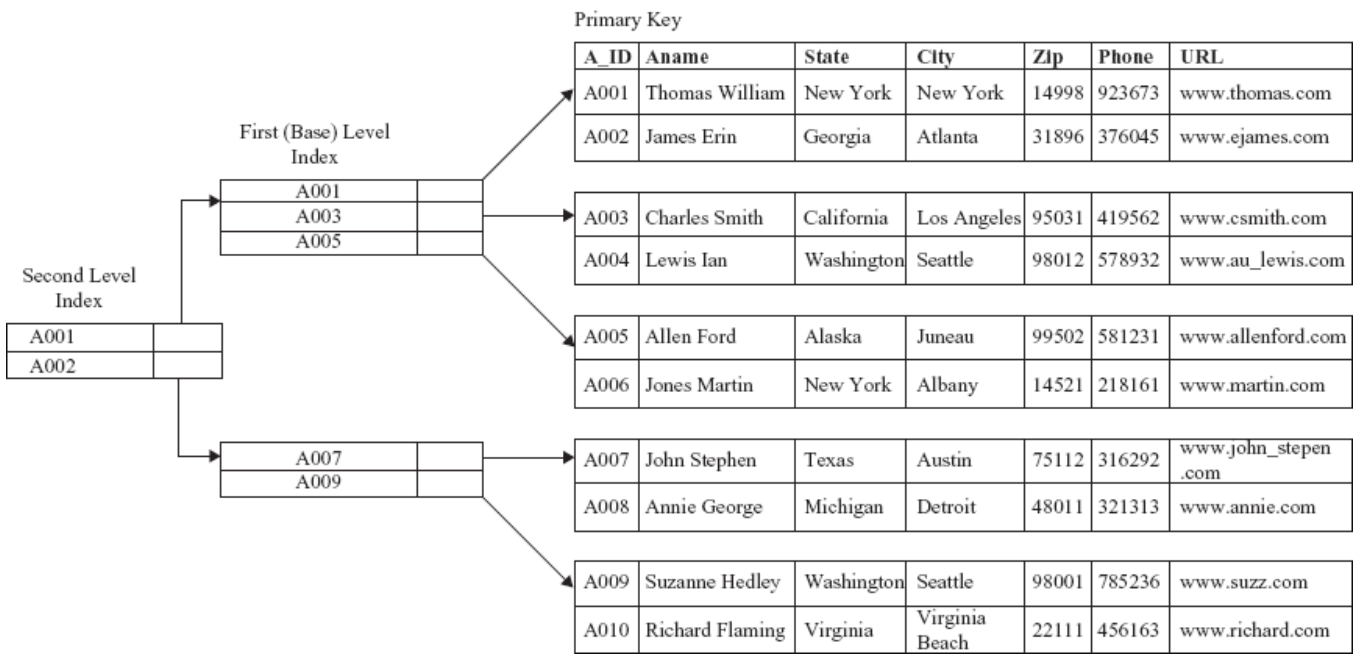
In general, secondary index occupies more space and consumes more time in searching, since it contains more entries than the primary index. However, having a secondary index is advantageous because in the absence of secondary index, accessing any random record requires a linear scan of file. On the other hand, if primary index does not exist, binary search is still applicable on the file, since records are ordered.

### 7.7.2 Multilevel Indexes

It is common to have several thousands (or even lacs) of records in the database of medium or large-scale organizations. As the size of the file grows, the size of the index (even sparse index) grows as well. If the size of the index becomes too large to fit in the main memory at once, it becomes inefficient for processing, since it must be kept sequentially on disk just like an ordinary file. It implies that searching large indexes require several disk-block accesses.

One solution to this problem is to view the index file, which we refer to as the **first** or **base level** of a multilevel index, just as any other sequential file and construct a primary index on the index file. This index to the first level is called the **second level** of the multilevel index. In order to search a record, we first apply binary search on the second level index to find the largest value which is less than or equal to the search-key. The pointer corresponding to this value points to the block of the first level index that contains an entry for the searched record. This block of first level index is searched to locate the largest value which is less than or equal to the search-key. The pointer corresponding to this value directs us to the block of file that contains the required record.

If second level index is too small to fit in main memory at once, fewer blocks of index file needs to be accessed from disk to locate a particular record. However, if the second level index file is too large to fit in main memory at once, another level of index can be created. In fact, this process of creating the index on index can be repeated until the size of index becomes small enough to fit in main memory. This type of index with multiple levels (two or more levels) of index is called **multilevel index**. Figure 7.24 illustrates multilevel indexing.



| A_ID | Aname | State | City | Zip | Phone | URL |
|---|---|---|---|---|---|---|
| A001 | Thomas William | New York | New York | 14998 | 923673 | www.thomas.com |
| A002 | James Erin | Georgia | Atlanta | 31896 | 376045 | www.ejames.com |
| A003 | Charles Smith | California | Los Angeles | 95031 | 419562 | www.csmith.com |
| A004 | Lewis Ian | Washington | Seattle | 98012 | 578932 | www.au_lewis.com |
| A005 | Allen Ford | Alaska | Juneau | 99502 | 581231 | www.allenford.com |
| A006 | Jones Martin | New York | Albany | 14521 | 218161 | www.martin.com |
| A007 | John Stephen | Texas | Austin | 75112 | 316292 | www.john_stepen .com |
| A008 | Annie George | Michigan | Detroit | 48011 | 321313 | www.annie.com |
| A009 | Suzanne Hedley | Washington | Seattle | 98001 | 785236 | www.suzz.com |
| A010 | Richard Flaming | Virginia | Virginia Beach | 22111 | 456163 | www.richard.com |

**Fig. 7.24**

Multilevel indexing reduces the number of disk accesses while searching for a record; however, insertion and deletion are complicated because all the index files at different levels are ordered files. Any insertion or deletion may cause several index files at different levels to be modified. One solution to this problem is to keep some space reserved in each block for new entries. This is often implemented using B-trees and B$^+$ -trees.

### *B-trees*

It is a kind of tree satisfying some additional properties or constraints. Each node of a B-tree consists of $n-1$ values and $n$ pointers. Pointers can be of two types, namely, *tree pointers* and *data pointers*. The tree pointer points to any other node of the tree and the data pointer points either to the block which contains the record with that index value or to the record itself. The order of information within each node is $<P_1, <V_1, R_1>, P_2, <V_2, R_2>, P_3, <V_3, R_3>, \ldots, P_{n-1}, <V_{n-1}, R_{n-1}>, P_n>$ where $P_i$ is a tree pointer, $V_i$ is any value of indexing attribute and $R_i$ is a data pointer (see Figure 7.25).

**Fig. 7.25** *A node of B-tree*

*NOTE* The structure of leaf node and internal node are same except the difference that all the tree pointers in a leaf node are null.

All the values in a node should be in the sorted order, it means $v_1 < v_2 < \cdots < v_{n-1}$ within each node of the tree. In addition, the node pointed to by any tree pointer $P_i$ should contain all the values larger than the value $v_{i-1}$ and less than the value $v_i$. Another constraint that the B-tree must satisfy is that it must be balanced; it means all the leaf nodes should be at the same level. B-tree also ensures that each node, except root and leaf nodes, should be at least half full, that means it should have at least $n/2$ tree pointers.

The number of tree pointers in each node (called **fan-out**) identifies the order of B-tree.

Maintaining all the constraints of a B-tree while insertion and deletion operation requires complex algorithms. First, consider the insertion of a value in a B-tree. Initially, the B-tree has only one node (root node) which is a leaf node at level 0. Suppose that the order of the B-tree is $n$. Thus, when $n-1$ values are inserted into the root node, it becomes full. Next insertion splits the root node into two nodes at level 1. The middle value, say $m$, is kept in the root node and the values from 1 to $m-1$ are shifted in the left child node and values from $m+1$ to $n-1$ are shifted in the right child node. However, when such situation occurs with a non-root node, it is split into two nodes at the same level and middle value is transferred to its parent node. Parent node then points to both the nodes as its left child and right child. If the parent node is also full, it is also split and splitting may propagate to other tree levels. If splitting gets propagated all the way to the root node and if root is also split, it increases the number of levels of the B-tree. Although, insertion in a node that is not full is quite simple and efficient. A typical B-tree of order $n = 3$ is shown in Figure 7.26.

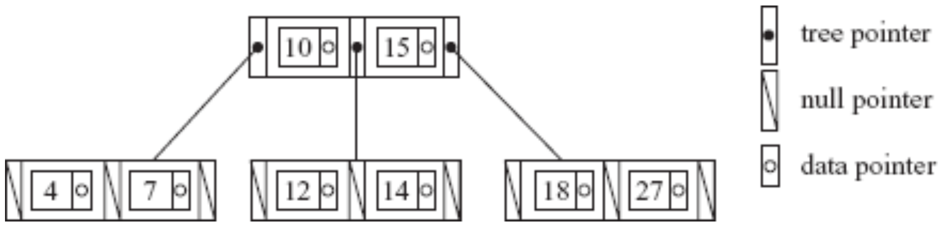
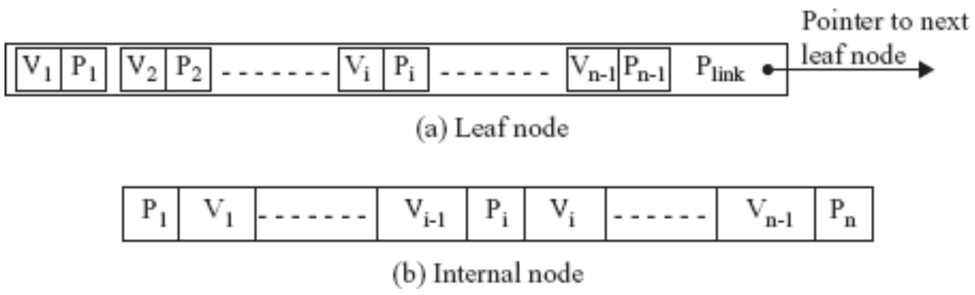


**Fig. 7.26** *A B-tree of order* $n=3$

Now, consider the deletion of a value from a B-tree. If deleting a value from a node leaves the node more than half empty, its values are merged with its neighbouring nodes. This may also propagate all the way to the root of the B-tree. Thus, deletion of a value may result in a tree with lesser number of levels than the number of levels of the tree before deletion. However, deletion from a node that is full or nearly full is a simple process.

## *B⁺-trees*

B⁺-tree, a variation of B-tree, is the most widely used structure for multilevel indexes. Unlike B-tree, B⁺-tree includes all the values of indexing attribute in the leaf nodes, and internal nodes contain values just for directing the search operation. All the pointers in the internal nodes are tree pointers and there is no data pointer in them. All the data pointers are included in the leaf nodes and there is one tree pointer in each leaf node that points to the next leaf node. Structure of leaf node and internal nodes of a B⁺-tree is shown in Figure 7.27.


$V_1$ $P_1$ $V_2$ $P_2$ $V_i$ $P_i$ $V_{n-1}$ $P_{n-1}$ $P_{link}$
Pointer to next leaf node
(a) Leaf node
$P_1$ $V_1$ $V_{i-1}$ $P_i$ $V_i$ $V_{n-1}$ $P_n$
(b) Internal node
**Fig. 7.27** *Nodes of a B⁺-tree*

Each leaf node has $n-1$ values along with a data pointer for each value. If the indexing attribute is the key attribute, the data pointer points either to the record in the file or to the disk block containing the record with that indexing attribute value. On the other hand, if the indexing attribute is not the key attribute, the data pointer points to the bucket of pointers, each of which points to the record in the file.

The leaf nodes and internal nodes of a B$^+$-tree corresponds to the first level and other levels of multilevel index, respectively.

Since data pointers are not included in the internal nodes of a B$^+$-tree, more entries can be fit into an internal node of a B$^+$-tree than corresponding B-tree. As a result, we have larger order for B$^+$-tree than B-tree for the same size of disk block. A typical B$^+$-tree of order $n=3$ is shown in Figure 7.28.

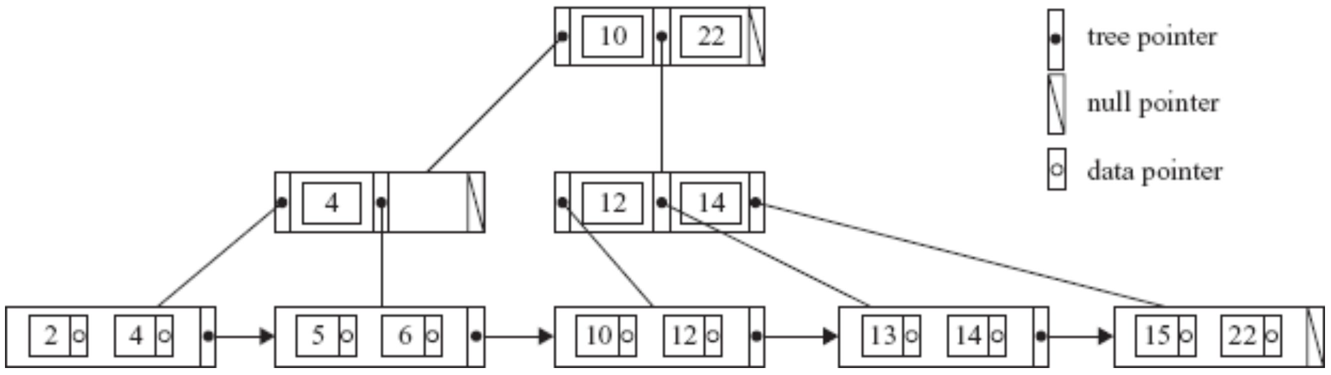


**Fig. 7.28** *A B$^+$-tree of order* $n=3$

In B-tree and B$^+$-tree data structure, each node corresponds to a disk block. In addition, each node is kept between half-full or completely full.

### *Indexed Sequential Access Method*

A variant of B$^+$-tree is **indexed sequential access method (ISAM)**. In ISAM, the leaf nodes and the overflow nodes (chained to some leaf nodes) contain all the values of indexing attribute and data pointers, and the internal nodes contain values and tree pointers just for directing the search operation. Unlike B$^+$-tree, the ISAM index structure is static, it means the number of leaf nodes never change; if need arises overflow node is allocated and chained to the leaf node. The leaf nodes are assumed to be allocated sequentially, thus, no pointer is needed in the leaf node to point to the next leaf node.

The algorithms for insertion, deletion, and search are simple. All searches start at root node and the value in the node helps in determining the next node to search. To insert a value, first the appropriate node is determined and if there is space in the node, the value is inserted there. Otherwise, an over-flow node is allocated and chained to the node, and then the value is inserted in the overflow node. To delete a value, the node containing the value is determined and the value is deleted from that node. In case, it is the only value in the overflow node, the overflow node is also removed. In case, it is the only value in the leaf node, the node is left empty to handle future insertions (data from overflow nodes, if any, linked to that leaf node is not moved to the leaf node).

The ISAM structure provides the benefits of sequentially allocated leaf nodes and fast searches as in the case of $B^+$-trees. However, the structure is unsuitable for situations where file grows or shrinks much. This is because if too much insertions are made to the same leaf node, a long overflow chain can develop, which degrades the performance while searching as overflow nodes need to be searched if the search reaches to that leaf node. A sample ISAM tree is shown in Figure 7.29.

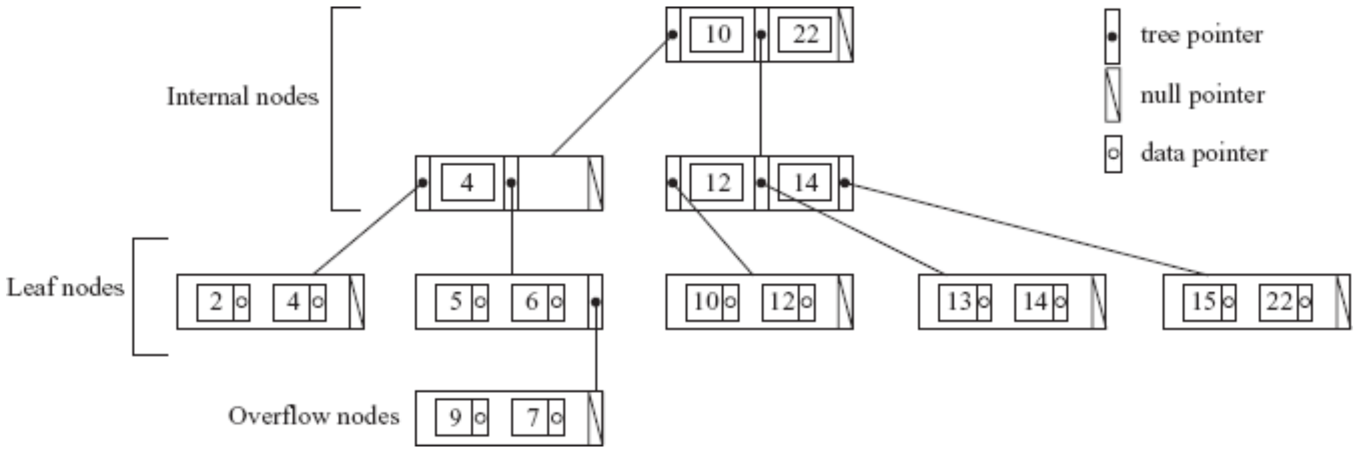


**Fig. 7.29** *Sample ISAM tree*

### 7.7.3 Indexes on Multiple Keys

So far, we have discussed that the index is created on primary or secondary keys, which consists of single attributes. However, many query requests are based on multiple attributes. For such queries, it is advantageous to create an index on multiple attributes that are involved in most of the queries.

To understand the concept, consider this query: List all the books from `BOOK` relation where `Category = "Textbook"` and `P_ID = "P001"`. This query involves two non-key attributes, namely, `Category` and `P_ID`, thus, multiple records may satisfy either condition. There are three possible strategies to process this query, which are given here.

- Assume that we have an index on `Category` field. Using this index we can find all the books having `Category = "Textbook"`. From these records, we can search all those books whose `P_ID = "P001"`.
- Assume that we have an index on `P_ID` field. Using this index we can find all the records having `P_ID = "P001"`. From these records, we can search all those books whose `Category = "Textbook"`.
- Assume that we have an index on both the attributes, namely, `Category` and `P_ID`. Using index on `Category` field, we can find the set of all the books whose `Category = "Textbook"`. Similarly, using index on `P_ID` field we can find all the books whose `P_ID = "P001"`. By performing the intersection of both the set, we can find all the books satisfying both the given conditions.

All of the strategies yield a correct set of records; however, if many records satisfy individual condition and only a few records satisfy both the conditions, then none of the above alternative is a good choice in terms of efficiency. Thus, some efficient technique is needed.

There are a number of techniques that would treat the combination of `Category` and `P_ID` as a search-key. One solution to this problem is an ordered index on multiple attributes. The idea behind **ordered index on multiple attribute** is to create an index on combination of more than one attribute. It means search-key consists of multiple attributes. Such a search-key containing multiple attribute is termed as **composite search-key**. In this example, search-key consists of both the attributes, namely, `Category` and `P_ID`. In this type of index, search-key represents a tuple with values $<v_1, v_2, v_3, \ldots, v_n>$ where index attributes are $<A_1, A_2, A_3,$

…, `A`$_n$>. The values of search-key are **lexicographic ordered**. Lexicographic ordering for above case states that all the entries of books published by *P001* precedes the books published by *P002* and so on. This type of index differs from the other index only in terms of search-key. However, basic structure is similar as that of any other index having search-key on single attribute. Thus, working of index on composite search-key is similar as any other index discussed so far.

Another solution to this problem is **partitioned hashing**, an extension of static hashing, which allows access on multiple keys. Partitioned hashing does not support range queries; however, it is suitable for equality comparisons. In partitioned hashing, for a composite search-key consisting of *n* attributes, the chosen hash function produces *n* separate hash addresses. These *n* addresses are concatenated to obtain the bucket address. All the buckets, which match the part of the address, are then searched for the combined search-key.

The main advantage of partitioned hashing is that it can be easily extended to any number of attributes in the search-key. In addition, there is no need to maintain separate access structure for individual attributes. On the other hand, the main disadvantage of partitioned hashing is that it does not support range queries on any of the component attributes.

Another alternative to such type of a problem is grid files. In the **grid files**, a grid array with one linear scale (or dimension) for each of the search attribute is created. Linear scales are made in order to provide uniform distribution of that attribute values in the grid array. Linear scale groups the values of one search-key attribute to each dimension. Figure 7.30 shows a grid array along with two linear scales, one is for attribute `Category` and the other is for attribute `P_ID`. Each grid cell points to some bucket address where the corresponding records are stored. The number of attributes in search-key determines the dimension of grid array. Thus, for search-key having *n* attributes, the grid array would have *n* dimensions. In our example, two attributes are taken as search-key, thus, the grid array is of two dimensions. Linear scale is looked up to map into the corresponding grid cell. Thus, the query for `Category` = “`Textbook`” and `P_ID` = “`P001`” maps into the cell (2, 0), which further points to the bucket address where the required records are stored.

The main advantage of this method is that it can be applied for range queries. In addition, it is extended to any number of attributes in the search-key. Grid file method is good for multiple-key search. On the other hand, the main disadvantage is that it represents space and management overhead in terms of grid array because it requires frequent reorganization of grid files with dynamic files.

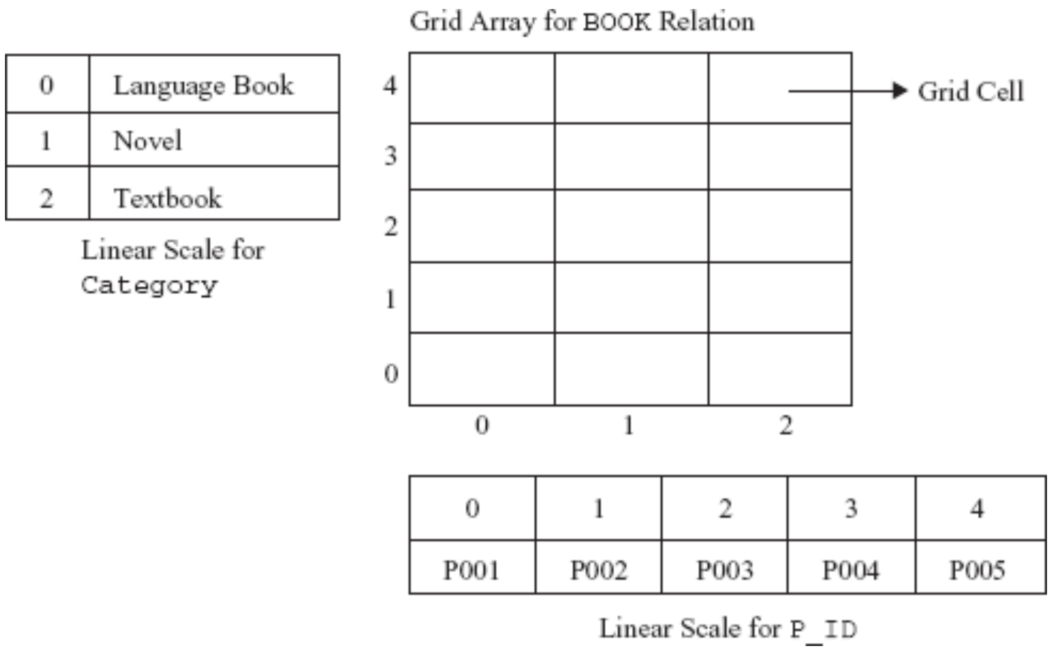


**Fig. 7.30** *Grid array along with two linear scales*

## SUMMARY

1. The database is stored on storage medium in the form of files, which is a collection of records consisting of related data items, such as name, address, phone, email-id, and so on.
2. Several types of storage media exist in computer systems. They are classified on the basis of speed with which they can access data. Among the media available are cache memory, main memory, flash memory, magnetic disks, optical disks, and magnetic tapes. Database is typically stored on the magnetic disk because of its higher capacity and persistent storage.

3. The storage media that consist of high-speed devices are referred to as primary storage. The storage media that consist of slower devices are referred to as secondary storage. Magnetic disk (generally called disk) is the primary form of secondary storage that enables storage of enormous amount of data. The slowest media comes under the category of tertiary storage. Removable media, such as optical discs and magnetic tapes, are considered as tertiary storage.
4. The CPU does not directly access data on the secondary storage and tertiary storage. Instead, data to be accessed must move to main memory so that it can be accessed.
5. A major advancement in secondary storage technology is represented by the development of Redundant Arrays of Independent Disks (RAID). The idea behind RAID is to have a large array of small independent disks. Data striping is used to improve the performance of disk, which utilizes parallelism. Mirroring (also termed as shadowing) is a technique used to improve the reliability of the disk. In this technique, the data is redundantly stored on two physical disks.
6. Several different RAID organizations are possible, each with different cost, performance, and reliability characteristics. Raid level 0 to 6 exists. RAID level 1 (mirroring) and RAID level 5 are the most commonly used.
7. In Storage Area Network (SAN) architecture, many server computers are connected with a large number of disks on a high-speed network. Storage disks are placed at a central server room, and are monitored and maintained by system administrators. An alternative to SAN is Network-Attached Storage (NAS). NAS device is a server that allows file sharing.
8. The database is mapped into a number of different files, which are physically stored on disk for persistency. Each file is decomposed into equal size pages, which is the unit of exchange between the disk and main memory.
9. It is not possible to keep all the pages of a file in main memory at

once; thus, the available space of main memory should be managed efficiently.

10. The main memory space available for storing the data is called buffer pool and the subsystem that manages the allocation of buffer space is called buffer manager.
11. Buffer manager uses replacement policy to choose a page for replacement if there is no space available for a new page in the main memory. The commonly used policies include LRU, MRU, and clock replacement policy.
12. There are two approaches to organize the database in the form of files. In the first approach, all the records of a file are of fixed-length. However, in the second approach, the records of a file vary in size.
13. Once the database is mapped to files, the records of these files can be mapped on to disk blocks. The two ways to organize them on disk blocks are spanned organization and unspanned organization.
14. There are various methods of organizing the records in a file while storing a file on disk. Some popular methods are heap file organization, sequential file organization, and hash file organization.
15. Heap file organization is the simplest file organization technique in which no particular order of record is maintained in the file. In sequential file, the records are physically placed in the sorted order based on the value of one or more fields, called the ordering field. The ordering field may or may not be the key field of the file, whereas records are organized using a technique called hashing in a hash file organization.
16. Hashing provides very fast access to an arbitrary record of a file, on the basis of certain search conditions. The hashing scheme in which a fixed number of buckets, say `N`, are allocated to a file to store records is called static hashing. Dynamic hashing technique allocates new buckets dynamically as needed. It allows the hash function to be modified dynamically to accommodate the growth or shrinkage of database files. Two hashing techniques for files that grow and shrink in number of records dynamically—namely, extendible and linear hashing.

17. To speed up the retrieval of records, additional access structures called indexes are used. Index can be created or defined on any field termed as indexing field or indexing attribute of the file. Creating single-level indexes, multilevel indexes, and indexes on multiple keys are possible.
18. Different types of single-level index are primary index, clustering index, and secondary index.
19. A primary index is an ordered file that contains an index on the primary key of the file. An index defined on a sequentially ordered non-key attribute (containing duplicate values) of a file is called clustering index. The attribute on which the clustering index is defined is known as a clustering attribute.
20. A secondary index is an index defined on a non-ordered attribute of the file. The indexing attribute may be a candidate key or a non-key attribute of the file.
21. The type of index with multiple levels (two or more levels) of index is called multilevel index. Multilevel indexes can be implemented using B-trees and $B^+$-trees, which are dynamic structures that allow an index to expand and shrink dynamically. $B^+$-trees can generally hold more entries in their internal nodes than B-trees.
22. Partitioned hashing is an extension of static hashing, which allows access on multiple keys. It does not support range queries; however, it is suitable for equality comparisons.

**KEY TERMS**

- Storage devices
- Volatile storage
- Non-volatile storage
- Primary storage
- Cache memory
- Main memory
- Flash memory
- Secondary storage
- Tertiary storage

- Optical disc
- CD-ROM
- DVD-ROM
- Tape storage
- Sequential access
- Direct access
- Magnetic disk
- Platter
- Single-sided disk
- Double-sided disk
- Tracks and sectors
- Read/write head
- Disk arm
- Spindle
- Head-disk assemblies
- Cylinder
- Disk controller
- SCSI, ATA, and SATA interfaces
- Checksums
- Seek time
- Rotational delay
- Data transfer time
- Access time
- Disk reliability and MTTF
- RAID
- Data striping
- Bit-level data striping
- Block-level data striping
- Redundant information
- Mirroring (or shadowing)
- Mean time to failure
- Mean time to repair
- Error correcting codes and check disk
- RAID levels

- Storage area network (SAN)
- Network-attached storage (NAS)
- Pages
- Buffer pool
- Buffer manager
- Frames
- Pinning and unpinning pages
- Page-replacement policies
- Least recently used
- Most recently used
- Clock replacement
- Fixed-length records
- File header
- Free list
- Variable-length records
- Mapping records
- Spanned organization
- Unspanned organization
- Slotted page structure
- File organization
- Heap file organization
- Heap file or pile file
- Sequential file organization
- Sequential file
- Ordered field
- Ordering key
- Hash file organization
- Hashing
- Hash file or direct file
- Hash function
- Hash table
- Cut key hashing
- Folded key hashing
- Division-remainder hashing

- Collision
- Collision resolution
- Open addressing
- Multiple addressing
- Chained overflow
- Bucket
- Static hashing
- Primary and overflow pages
- Overflow chain of bucket
- Dynamic hashing
- Extendible hashing
- Directory of pointers
- Global depth
- Local depth
- Linear hashing
- Indexing
- Index
- Indexing field or indexing attribute
- Single-level index
- Primary index
- Sparse index
- Dense index
- Clustering index
- Clustering attribute
- Secondary index
- Multilevel index
- First or base level
- Second level
- B-tree
- Tree pointers
- Data pointers
- $B^+$-tree
- Indexes on multiple keys
- Composite search-key

- Lexicographic order
- Partitioned hashing
- Grid files

**EXERCISES**

**A. Multiple Choice Questions**

1. Which of the following is not classified as primary storage?
    1. Flash memory
    2. Main memory
    3. Magnetic disk
    4. Cache
2. Which of the following term is not related to magnetic disk?
    1. Arm
    2. Cylinder
    3. Platter
    4. None of these
3. The time taken to position read/write heads on specific track is known as ________.
    1. Rotational delay time
    2. Seek time
    3. Data transfer time
    4. Access time
4. Which of the following RAID level uses block-level striping?
    1. Level 0
    2. Level 1
    3. Level 4
    4. Level 6
5. ________ actually handles the request to access a page of a file.
    1. Disk controller
    2. Database
    3. Buffer manager
    4. Main memory
6. Which replacement policy requires the reference bit to be associated

with each frame?
   1. Least recently used
   2. Most recently used
   3. Clock replacement
   4. Both (a) and (b)
7. While mapping fixed-length records to disk, which of the following organization is generally used?
   1. Spanned organization
   2. Unspanned organization
   3. Both (a) and (b)
   4. None of these
8. Which of the following is not a file organization method?
   1. Heap file organization
   2. Sequential file organization
   3. Hash file organization
   4. None of these
9. Which of the following hash functions never result in collision?
   1. Cut key
   2. Folded key
   3. Division-remainder
   4. None of these
10. The type of indexes that include an entry for each value of indexing attribute is known as ________.
    1. Clustering index
    2. Sparse index
    3. Dense index
    4. Secondary index

**B. Fill in the Blanks**

1. _________ is a static RAM and is very small in size.
2. Disk surface of a platter is divided into imaginary ________ and _______.
3. The reliability of the disk is measured in terms of ___________.
4. In order to improve the performance of disk, a concept called

________ is used which utilizes parallelism.

5. In practice, the RAID level ________ is not used.
6. The process of incrementing the `pin_count` is generally called ________ the page.
7. A large number of database systems use ________ page replacement policy.
8. One common technique to implement variable-length record is ______________.
9. A file organization in which records are sorted based on the value of one of its field is called ______________.
10. In static hashing, the pages of a file can be viewed as a collection of buckets, with one ________ page and additional __________ pages.

**C. Answer the Questions**

1. List the different types of storage media available in computer system. Also explain how they are classified into different categories.
2. Give hardware description and various features of magnetic disk. How do you measure its performance?
3. Define these terms: buffer manager, buffer pool, mirroring, file header, free list, ordering field, ordering key.
4. Define RAID. What is the need of having RAID technology?
5. How can the reliability and performance of disk be improved using RAID? Explain different RAID levels.
6. Define the role of buffer manager. Explain the policies used by buffer manager to replace a page.
7. Compare and contrast fixed-length and variable-length records. Discuss various techniques that the system uses to delete a record from a file.
8. Why do variable-length records require separator characters? Is there any alternative of using separator character? Justify your answer.
9. Compare and contrast spanned and unspanned organization of records. What is slotted page structure, and why is it needed?
10. Discuss the importance of file organization in databases. Define

various types of file organizations available.

11. What is page replacement policy? Compare and contrast LRU and MRU replacement policies with the help of example.
12. Discuss the two techniques of implementing heap file with diagram.
13. Discuss the commonly used hash functions. What is the main problem associated with most of them, and how can it be resolved?
14. How does hashing allow a file to expand and shrink dynamically?
15. What are the advantages of having an index on a file? List the different types of single-level indexes available. Discuss them in detail.
16. What are dense and sparse indexes?
17. What is multilevel index? Also give reasons for having multilevel index.
18. What does the order of a B-tree and $B^+$-tree indicate? Also explain the structure of both internal and external nodes of a B-tree and $B^+$-tree.
19. Why is $B^+$-tree preferable over B-tree?
20. Explain the significance of index defined on multiple attributes. Also explain partitioned hashing and grid files with example.
21. Write a short note on the following.
    1. Clock replacement policy
    2. SAN and NAS
    3. Partitioned hashing

### D. Practical Questions

1. Consider an empty $B^+$-tree with order `p=3`, and perform the following.
    1. Show the tree after inserting these values: 70, 15, 20, 35, 18, 55, 43. Assume that the values are inserted in the order given
    2. Show the tree after deleting the node with value 18 from the tree
    3. Show the steps involved in finding the value 35
2. Perform (a), (b), and part of previous question for a B-tree with order `p=3`.

3. Create a linear hash file on a file that contains records with these key values: 70, 15, 20, 35, 18, 55, 43. Suppose the hash function used is `h(k) = k mod 5`, and each bucket can hold three records.
4. Show the hash file created in question 3 when following steps are performed on the database file.
   1. Insertion of a record with key value 12
   2. Deletion of record with key value 18
5. Consider a fixed-length record file having 1500 records of `PUBLISHER` relation (defined in Section 7.5.1). Assume that block size is 100 bytes. How many blocks are needed to store the file if records are organized using
   1. spanned organization, assuming each block has a 5-byte pointer to the next block
   2. unspanned organization

   Calculate the number of bytes wasted in each block due to unspanned organization