CHAPTER 10

# Storage and File Structure

In preceding chapters, we have emphasized the higher-level models of a database. For example, at the *conceptual* or *logical* level, we viewed the database, in the relational model, as a collection of tables. Indeed, the logical model of the database is the correct level for database *users* to focus on. This is because the goal of a database system is to simplify and facilitate access to data; users of the system should not be burdened unnecessarily with the physical details of the implementation of the system.

In this chapter, however, as well as in Chapters 11, 12, and 13, we probe below the higher levels as we describe various methods for implementing the data models and languages presented in preceding chapters. We start with characteristics of the underlying storage media, such as disk and tape systems. We then define various data structures that allow fast access to data. We consider several alternative structures, each best suited to a different kind of access to data. The final choice of data structure needs to be made on the basis of the expected use of the system and of the physical characteristics of the specific machine.

## 10.1 Overview of Physical Storage Media

Several types of data storage exist in most computer systems. These storage media are classified by the speed with which data can be accessed, by the cost per unit of data to buy the medium, and by the medium's reliability. Among the media typically available are these:

- **Cache**. The cache is the fastest and most costly form of storage. Cache memory is relatively small; its use is managed by the computer system hardware. We shall not be concerned about managing cache storage in the database system. It is, however, worth noting that database implementors do pay attention to cache effects when designing query processing data structures and algorithms.
- **Main memory**. The storage medium used for data that are available to be operated on is main memory. The general-purpose machine instructions operate

on main memory. Although main memory may contain several gigabytes of data on a personal computer, or even hundreds of gigabytes of data in large server systems, it is generally too small (or too expensive) for storing the entire database. The contents of main memory are usually lost if a power failure or system crash occurs.

- **Flash memory**. Flash memory differs from main memory in that stored data are retained even if power is turned off (or fails). There are two types of flash memory, called *NAND* and *NOR* flash. Of these, NAND flash has a much higher storage capacity for a given cost, and is widely used for data storage in devices such as cameras, music players, and cell phones, and increasingly, in laptop computers as well. Flash memory has a lower cost per byte than main memory, in addition to being nonvolatile; that is, it retains stored data even if power is switched off.

  Flash memory is also widely used for storing data in "USB keys," which can be plugged into the **Universal Serial Bus** (USB) slots of computing devices. Such USB keys have become a popular means of transporting data between computer systems ("floppy disks" played the same role in earlier days, but their limited capacity has made them obsolete now).

  Flash memory is also increasingly used as a replacement for magnetic disks for storing moderate amounts of data. Such disk-drive replacements are called *solid-state drives*. As of 2009, a 64 GB solid-state hard drive costs less than $200, and capacities range up to 160 GB. Further, flash memory is increasingly being used in server systems to improve performance by caching frequently used data, since it provides faster access than disk, with larger storage capacity than main memory (for a given cost).

- **Magnetic-disk storage**. The primary medium for the long-term online storage of data is the magnetic disk. Usually, the entire database is stored on magnetic disk. The system must move the data from disk to main memory so that they can be accessed. After the system has performed the designated operations, the data that have been modified must be written to disk.

  As of 2009, the size of magnetic disks ranges from 80 gigabytes to 1.5 terabytes, and a 1 terabyte disk costs about $100. Disk capacities have been growing at about 50 percent per year, and we can expect disks of much larger capacity every year. Disk storage survives power failures and system crashes. Disk-storage devices themselves may sometimes fail and thus destroy data, but such failures usually occur much less frequently than do system crashes.

- **Optical storage**. The most popular forms of optical storage are the *compact disk* (CD), which can hold about 700 megabytes of data and has a playtime of about 80 minutes, and the *digital video disk* (DVD), which can hold 4.7 or 8.5 gigabytes of data per side of the disk (or up to 17 gigabytes on a two-sided disk). The expression **digital versatile disk** is also used in place of **digital video disk**, since DVDs can hold any digital data, not just video data. Data are stored optically on a disk, and are read by a laser. A higher capacity format called *Blu-ray DVD* can store 27 gigabytes per layer, or 54 gigabytes in a double-layer disk.

The optical disks used in read-only compact disks (CD-ROM) or read-only digital video disks (DVD-ROM) cannot be written, but are supplied with data prerecorded. There are also "record-once" versions of compact disk (called CD-R) and digital video disk (called DVD-R and DVD+R), which can be written only once; such disks are also called **write-once, read-many** (WORM) disks. There are also "multiple-write" versions of compact disk (called CD-RW) and digital video disk (DVD-RW, DVD+RW, and DVD-RAM), which can be written multiple times.

Optical disk **jukebox** systems contain a few drives and numerous disks that can be loaded into one of the drives automatically (by a robot arm) on demand.

- **Tape storage**. Tape storage is used primarily for backup and archival data. Although magnetic tape is cheaper than disks, access to data is much slower, because the tape must be accessed sequentially from the beginning. For this reason, tape storage is referred to as **sequential-access** storage. In contrast, disk storage is referred to as **direct-access** storage because it is possible to read data from any location on disk.

  Tapes have a high capacity (40- to 300-gigabyte tapes are currently available), and can be removed from the tape drive, so they are well suited to cheap archival storage. Tape libraries (jukeboxes) are used to hold exceptionally large collections of data such as data from satellites, which could include as much as hundreds of terabytes (1 terabyte = $10^{12}$ bytes), or even multiple petabytes (1 petabyte = $10^{15}$ bytes) of data in a few cases.

The various storage media can be organized in a hierarchy (Figure 10.1) according to their speed and their cost. The higher levels are expensive, but are fast. As we move down the hierarchy, the cost per bit decreases, whereas the access time increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and semiconductor memory have become faster and cheaper. Magnetic tapes themselves were used to store active data back when disks were expensive and had low storage capacity. Today, almost all active data are stored on disks, except in very rare cases where they are stored on tape or in optical jukeboxes.

The fastest storage media—for example, cache and main memory—are referred to as **primary storage**. The media in the next level in the hierarchy—for example, magnetic disks—are referred to as **secondary storage**, or **online storage**. The media in the lowest level in the hierarchy—for example, magnetic tape and optical-disk jukeboxes—are referred to as **tertiary storage**, or **offline storage**.

In addition to the speed and cost of the various storage systems, there is also the issue of storage volatility. **Volatile storage** loses its contents when the power to the device is removed. In the hierarchy shown in Figure 10.1, the storage systems from main memory up are volatile, whereas the storage systems below

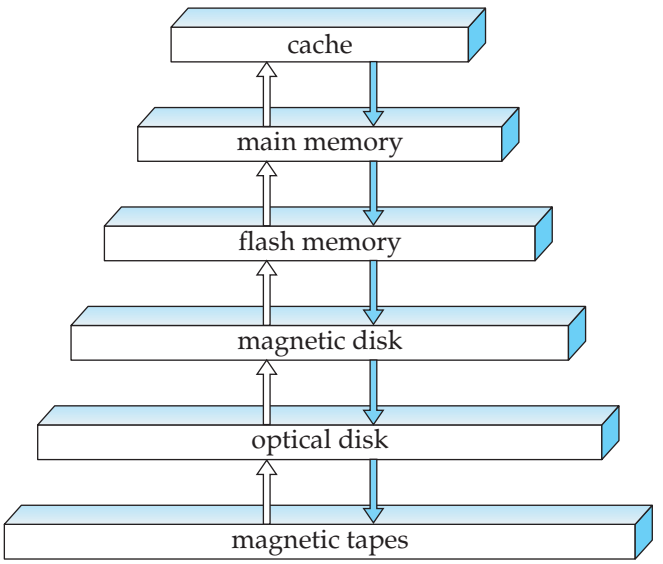



**Figure 10.1** Storage device hierarchy.

main memory are nonvolatile. Data must be written to **nonvolatile storage** for safekeeping. We shall return to this subject in Chapter 16.

## 10.2 Magnetic Disk and Flash Storage

Magnetic disks provide the bulk of secondary storage for modern computer systems. Although disk capacities have been growing year after year, the storage requirements of large applications have also been growing very fast, in some cases even faster than the growth rate of disk capacities. A very large database may require hundreds of disks. In recent years, flash-memory storage sizes have grown rapidly, and flash storage is increasingly becoming a competitor to magnetic disk storage for several applications.

### 10.2.1 Physical Characteristics of Disks

Physically, disks are relatively simple (Figure 10.2). Each disk **platter** has a flat, circular shape. Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces. Platters are made from rigid metal or glass.

When the disk is in use, a drive motor spins it at a constant high speed (usually 60, 90, or 120 revolutions per second, but disks running at 250 revolutions per second are available). There is a read–write head positioned just above the surface of the platter. The disk surface is logically divided into **tracks**, which are subdivided into **sectors**. A **sector** is the smallest unit of information that can be read from or written to the disk. In currently available disks, sector sizes are

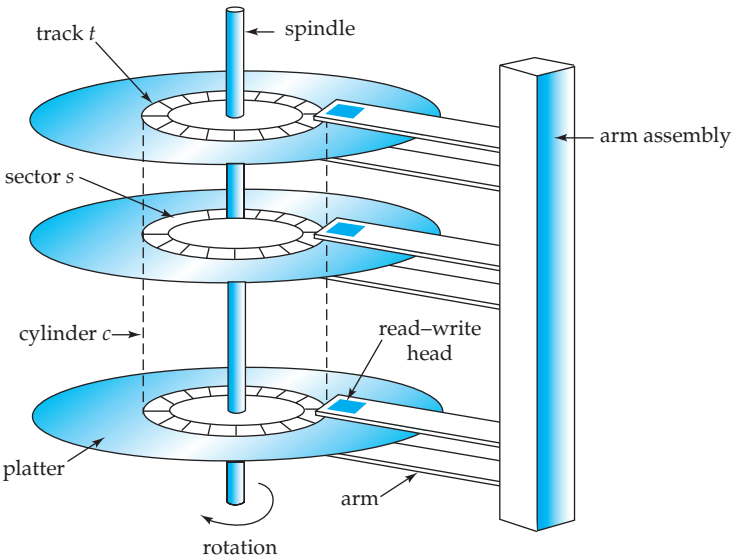



**Figure 10.2** Moving head disk mechanism.

typically 512 bytes; there are about 50,000 to 100,000 tracks per platter, and 1 to 5 platters per disk. The inner tracks (closer to the spindle) are of smaller length, and in current-generation disks, the outer tracks contain more sectors than the inner tracks; typical numbers are around 500 to 1000 sectors per track in the inner tracks, and around 1000 to 2000 sectors per track in the outer tracks. The numbers vary among different models; higher-capacity models usually have more sectors per track and more tracks on each platter.

The **read–write head** stores information on a sector magnetically as reversals of the direction of magnetization of the magnetic material.

Each side of a platter of a disk has a read–write head that moves across the platter to access different tracks. A disk typically contains many platters, and the read–write heads of all the tracks are mounted on a single assembly called a **disk arm**, and move together. The disk platters mounted on a spindle and the heads mounted on a disk arm are together known as **head–disk assemblies**. Since the heads on all the platters move together, when the head on one platter is on the $i$th track, the heads on all other platters are also on the $i$th track of their respective platters. Hence, the $i$th tracks of all the platters together are called the $i$th **cylinder**.

Today, disks with a platter diameter of $3\frac{1}{2}$ inches dominate the market. They have a lower cost and faster seek times (due to smaller seek distances) than do the larger-diameter disks (up to 14 inches) that were common earlier, yet they provide high storage capacity. Disks with even smaller diameters are used in portable devices such as laptop computers, and some handheld computers and portable music players.

The read–write heads are kept as close as possible to the disk surface to increase the recording density. The head typically floats or flies only microns

from the disk surface; the spinning of the disk creates a small breeze, and the head assembly is shaped so that the breeze keeps the head floating just above the disk surface. Because the head floats so close to the surface, platters must be machined carefully to be flat.

Head crashes can be a problem. If the head contacts the disk surface, the head can scrape the recording medium off the disk, destroying the data that had been there. In older-generation disks, the head touching the surface caused the removed medium to become airborne and to come between the other heads and their platters, causing more crashes; a head crash could thus result in failure of the entire disk. Current-generation disk drives use a thin film of magnetic metal as recording medium. They are much less susceptible to failure by head crashes than the older oxide-coated disks.

A **disk controller** interfaces between the computer system and the actual hardware of the disk drive; in modern disk systems, the disk controller is implemented within the disk drive unit. A disk controller accepts high-level commands to read or write a sector, and initiates actions, such as moving the disk arm to the right track and actually reading or writing the data. Disk controllers also attach **checksums** to each sector that is written; the checksum is computed from the data written to the sector. When the sector is read back, the controller computes the checksum again from the retrieved data and compares it with the stored checksum; if the data are corrupted, with a high probability the newly computed checksum will not match the stored checksum. If such an error occurs, the controller will retry the read several times; if the error continues to occur, the controller will signal a read failure.

Another interesting task that disk controllers perform is **remapping of bad sectors**. If the controller detects that a sector is damaged when the disk is initially formatted, or when an attempt is made to write the sector, it can logically map the sector to a different physical location (allocated from a pool of extra sectors set aside for this purpose). The remapping is noted on disk or in nonvolatile memory, and the write is carried out on the new location.

Disks are connected to a computer system through a high-speed interconnection. There are a number of common interfaces for connecting disks to computers of which the most commonly used today are (1) **SATA** (which stands for **serial ATA**,[1] and a newer version of SATA called SATA II or SATA3 Gb (older versions of the ATA standard called **PATA**, or Parallel ATA, and IDE, were widely used earlier, and are still available), (2) **small-computer-system interconnect** (SCSI; pronounced "scuzzy") , (3) SAS (which stands for **serial attached SCSI**), and (4) the Fibre Channel interface. Portable external disk systems often use the USB interface or the IEEE 1394 FireWire interface.

While disks are usually connected directly by cables to the disk interface of the computer system, they can be situated remotely and connected by a high-speed network to the disk controller. In the **storage area network** (**SAN**) architecture, large numbers of disks are connected by a high-speed network to a number

[1] ATA is a storage-device connection standard from the 1980s.

of server computers. The disks are usually organized locally using a storage organization technique called **redundant arrays of independent disks** (RAID) (described later, in Section 10.3), to give the servers a logical view of a very large and very reliable disk. The computer and the disk subsystem continue to use the SCSI, SAS, or Fiber Channel interface protocols to talk with each other, although they may be separated by a network. Remote access to disks across a storage area network means that disks can be shared by multiple computers that could run different parts of an application in parallel. Remote access also means that disks containing important data can be kept in a central server room where they can be monitored and maintained by system administrators, instead of being scattered in different parts of an organization.

**Network attached storage (NAS)** is an alternative to SAN. NAS is much like SAN, except that instead of the networked storage appearing to be a large disk, it provides a file system interface using networked file system protocols such as NFS or CIFS.

### 10.2.2 Performance Measures of Disks

The main measures of the qualities of a disk are capacity, access time, data-transfer rate, and reliability.

**Access time** is the time from when a read or write request is issued to when data transfer begins. To access (that is, to read or write) data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates. The time for repositioning the arm is called the **seek time**, and it increases with the distance that the arm must move. Typical seek times range from 2 to 30 milliseconds, depending on how far the track is from the initial arm position. Smaller disks tend to have lower seek times since the head has to travel a smaller distance.

The **average seek time** is the average of the seek times, measured over a sequence of (uniformly distributed) random requests. If all tracks have the same number of sectors, and we disregard the time required for the head to start moving and to stop moving, we can show that the average seek time is one-third the worst-case seek time. Taking these factors into account, the average seek time is around one-half of the maximum seek time. Average seek times currently range between 4 and 10 milliseconds, depending on the disk model.

Once the head has reached the desired track, the time spent waiting for the sector to be accessed to appear under the head is called the **rotational latency time**. Rotational speeds of disks today range from 5400 rotations per minute (90 rotations per second) up to 15,000 rotations per minute (250 rotations per second), or, equivalently, 4 milliseconds to 11.1 milliseconds per rotation. On an average, one-half of a rotation of the disk is required for the beginning of the desired sector to appear under the head. Thus, the **average latency time** of the disk is one-half the time for a full rotation of the disk.

The access time is then the sum of the seek time and the latency, and ranges from 8 to 20 milliseconds. Once the first sector of the data to be accessed has come under the head, data transfer begins. The **data-transfer rate** is the rate at which

data can be retrieved from or stored to the disk. Current disk systems support maximum transfer rates of 25 to 100 megabytes per second; transfer rates are significantly lower than the maximum transfer rates for inner tracks of the disk, since they have fewer sectors. For example, a disk with a maximum transfer rate of 100 megabytes per second may have a sustained transfer rate of around 30 megabytes per second on its inner tracks.

The final commonly used measure of a disk is the **mean time to failure** (MTTF), which is a measure of the reliability of the disk. The mean time to failure of a disk (or of any other system) is the amount of time that, on average, we can expect the system to run continuously without any failure. According to vendors' claims, the mean time to failure of disks today ranges from 500,000 to 1,200,000 hours—about 57 to 136 years. In practice the claimed mean time to failure is computed on the probability of failure when the disk is new—the figure means that given 1000 relatively new disks, if the MTTF is 1,200,000 hours, on an average one of them will fail in 1200 hours. A mean time to failure of 1,200,000 hours does not imply that the disk can be expected to function for 136 years! Most disks have an expected life span of about 5 years, and have significantly higher rates of failure once they become more than a few years old.

Disk drives for desktop machines typically support the Serial ATA(SATA) interface, which supports 150 megabytes per second, or the SATA-II 3Gb interface, which supports 300 megabytes per second. The PATA 5 interface supported transfer rates of 133 megabytes per second. Disk drives designed for server systems typically support the Ultra320 SCSI interface, which provides transfer rates of up to 320 megabytes per second, or the Serial Attached SCSI (SAS) interface, versions of which provide transfer rates of 3 or 6 gigabits per second. Storage area network (SAN) devices, which are connected to servers by a network, typically use Fiber Channel FC 2-Gb or 4-Gb interface, which provides transfer rates of up to 256 or 512 megabytes per second. The transfer rate of an interface is shared between all disks attached to the interface, except for the serial interfaces which allow only one disk to be connected to each interface.

### 10.2.3 Optimization of Disk-Block Access

Requests for disk I/O are generated both by the file system and by the virtual memory manager found in most operating systems. Each request specifies the address on the disk to be referenced; that address is in the form of a *block number*. A **block** is a logical unit consisting of a fixed number of contiguous sectors. Block sizes range from 512 bytes to several kilobytes. Data are transferred between disk and main memory in units of blocks. The term **page** is often used to refer to blocks, although in a few contexts (such as flash memory) they refer to different things.

A sequence of requests for blocks from disk may be classified as a sequential access pattern or a random access pattern. In a **sequential access** pattern, successive requests are for successive block numbers, which are on the same track, or on adjacent tracks. To read blocks in sequential access, a disk seek may be required for the first block, but successive requests would either not require a seek, or

require a seek to an adjacent track, which is faster than a seek to a track that is farther away.

In contrast, in a **random access** pattern, successive requests are for blocks that are randomly located on disk. Each such request would require a seek. The number of random block accesses that can be satisfied by a single disk in a second depends on the seek time, and is typically about 100 to 200 accesses per second. Since only a small amount (one block) of data is read per seek, the transfer rate is significantly lower with a random access pattern than with a sequential access pattern.

A number of techniques have been developed for improving the speed of access to blocks.

- **Buffering**. Blocks that are read from disk are stored temporarily in an in-memory buffer, to satisfy future requests. Buffering is done by both the operating system and the database system. Database buffering is discussed in more detail in Section 10.8.
- **Read-ahead**. When a disk block is accessed, consecutive blocks from the same track are read into an in-memory buffer even if there is no pending request for the blocks. In the case of sequential access, such read-ahead ensures that many blocks are already in memory when they are requested, and minimizes the time wasted in disk seeks and rotational latency per block read. Operating systems also routinely perform read-ahead for consecutive blocks of an operating system file. Read-ahead is, however, not very useful for random block accesses.
- **Scheduling**. If several blocks from a cylinder need to be transferred from disk to main memory, we may be able to save access time by requesting the blocks in the order in which they will pass under the heads. If the desired blocks are on different cylinders, it is advantageous to request the blocks in an order that minimizes disk-arm movement. **Disk-arm–scheduling** algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed. A commonly used algorithm is the **elevator algorithm**, which works in the same way many elevators do. Suppose that, initially, the arm is moving from the innermost track toward the outside of the disk. Under the elevator algorithm's control, for each track for which there is an access request, the arm stops at that track, services requests for the track, and then continues moving outward until there are no waiting requests for tracks farther out. At this point, the arm changes direction, and moves toward the inside, again stopping at each track for which there is a request, until it reaches a track where there is no request for tracks farther toward the center. Then, it reverses direction and starts a new cycle. Disk controllers usually perform the task of reordering read requests to improve performance, since they are intimately aware of the organization of blocks on disk, of the rotational position of the disk platters, and of the position of the disk arm.

- **File organization**. To reduce block-access time, we can organize blocks on disk in a way that corresponds closely to the way we expect data to be accessed. For example, if we expect a file to be accessed sequentially, then we should ideally keep all the blocks of the file sequentially on adjacent cylinders. Older operating systems, such as the IBM mainframe operating systems, provided programmers fine control on placement of files, allowing a programmer to reserve a set of cylinders for storing a file. However, this control places a burden on the programmer or system administrator to decide, for example, how many cylinders to allocate for a file, and may require costly reorganization if data are inserted to or deleted from the file.

  Subsequent operating systems, such as Unix and Microsoft Windows, hide the disk organization from users, and manage the allocation internally. Although they do not guarantee that all blocks of a file are laid out sequentially, they allocate multiple consecutive blocks (an **extent**) at a time to a file. Sequential access to the file then only needs one seek per extent, instead of one seek per block. Over time, a sequential file that has multiple small appends may become **fragmented**; that is, its blocks become scattered all over the disk. To reduce fragmentation, the system can make a backup copy of the data on disk and restore the entire disk. The restore operation writes back the blocks of each file contiguously (or nearly so). Some systems (such as different versions of the Windows operating system) have utilities that scan the disk and then move blocks to decrease the fragmentation. The performance increases realized from these techniques can be large.

- **Nonvolatile write buffers**. Since the contents of main memory are lost in a power failure, information about database updates has to be recorded on disk to survive possible system crashes. For this reason, the performance of update-intensive database applications, such as transaction-processing systems, is heavily dependent on the speed of disk writes.

  We can use **nonvolatile random-access memory** (NVRAM) to speed up disk writes drastically. The contents of NVRAM are not lost in power failure. A common way to implement NVRAM is to use battery–backed-up RAM, although flash memory is also increasingly being used for nonvolatile write buffering. The idea is that, when the database system (or the operating system) requests that a block be written to disk, the disk controller writes the block to an NVRAM buffer, and immediately notifies the operating system that the write completed successfully. The controller writes the data to their destination on disk whenever the disk does not have any other requests, or when the NVRAM buffer becomes full. When the database system requests a block write, it notices a delay only if the NVRAM buffer is full. On recovery from a system crash, any pending buffered writes in the NVRAM are written back to the disk. NVRAM buffers are found in certain high end disks, but are more frequently found in "RAID controllers"; we study RAID in Section 10.3.

- **Log disk**. Another approach to reducing write latencies is to use a log disk—that is, a disk devoted to writing a sequential log—in much the same way as a nonvolatile RAM buffer. All access to the log disk is sequential, essentially

eliminating seek time, and several consecutive blocks can be written at once, making writes to the log disk several times faster than random writes. As before, the data have to be written to their actual location on disk as well, but the log disk can do the write later, without the database system having to wait for the write to complete. Furthermore, the log disk can reorder the writes to minimize disk-arm movement. If the system crashes before some writes to the actual disk location have completed, when the system comes back up it reads the log disk to find those writes that had not been completed, and carries them out then.

File systems that support log disks as above are called **journaling file systems**. Journaling file systems can be implemented even without a separate log disk, keeping data and the log on the same disk. Doing so reduces the monetary cost, at the expense of lower performance.

Most modern file systems implement journaling, and use the log disk when writing internal file system information such as file allocation information. Earlier-generation file systems allowed write reordering without using a log disk, and ran the risk that the file system data structures on disk would be corrupted if the system crashed. Suppose, for example, that a file system used a linked list, and inserted a new node at the end by first writing the data for the new node, then updating the pointer from the previous node. Suppose also that the writes were reordered, so the pointer was updated first, and the system crashes before the new node is written. The contents of the node would then be whatever junk was on disk earlier, resulting in a corrupted data structure.

To deal with the possibility of such data structure corruption, earlier-generation file systems had to perform a file system consistency check on system restart, to ensure that the data structures were consistent. And if they were not, extra steps had to be taken to restore them to consistency. These checks resulted in long delays in system restart after a crash, and the delays became worse as disk systems grew to higher capacities. Journaling file systems allow quick restart without the need for such file system consistency checks.

However, writes performed by applications are usually not written to the log disk. Database systems implement their own forms of logging, which we study later in Chapter 16.

### 10.2.4 Flash Storage

As mentioned in Section 10.1, there are two types of flash memory, NOR flash and NAND flash. NOR flash allows random access to individual words of memory, and has read time comparable to main memory. However, unlike NOR flash, reading from NAND flash requires an entire *page* of data, typically consisting of between 512 and 4096 bytes, to be fetched from NAND flash into main memory. Pages in a NAND flash are thus similar to sectors in a magnetic disk. But NAND flash is significantly cheaper than NOR flash, and has much higher storage capacity, and is by far the more widely used.

Storage systems built using NAND flash provide the same block-oriented interface as disk storage. Compared to magnetic disks, flash memory can provide much faster random access: a page of data can be retrieved in around 1 or 2 microseconds from flash, whereas a random access on disk would take 5 to 10 milliseconds. Flash memory has a lower transfer rate than magnetic disks, with 20 megabytes per second being common. Some more recent flash memories have increased transfer rates of 100 to 200 megabytes per second. However, solid state drives use multiple flash memory chips in parallel, to increase transfer rates to over 200 megabytes per second, which is faster than transfer rates of most disks.

Writes to flash memory are a little more complicated. A write to a page of flash memory typically takes a few microseconds. However, once written, a page of flash memory cannot be directly overwritten. Instead, it has to be erased and rewritten subsequently. The erase operation can be performed on a number of pages, called an **erase block**, at once, and takes about 1 to 2 milliseconds. The size of an erase block (often referred to as just "block" in flash literature) is usually significantly larger than the block size of the storage system. Further, there is a limit to how many times a flash page can be erased, typically around 100,000 to 1,000,000 times. Once this limit is reached, errors in storing bits are likely to occur.

Flash memory systems limit the impact of both the slow erase speed and the update limits by mapping logical page numbers to physical page numbers. When a logical page is updated, it can be remapped to any already erased physical page, and the original location can be erased later. Each physical page has a small area of memory where its logical address is stored; if the logical address is remapped to a different physical page, the original physical page is marked as deleted. Thus by scanning the physical pages, we can find where each logical page resides. The logical-to-physical page mapping is replicated in an in-memory **translation table** for quick access.

Blocks containing multiple deleted pages are periodically erased, taking care to first copy nondeleted pages in those blocks to a different block (the translation table is updated for these nondeleted pages). Since each physical page can be updated only a fixed number of times, physical pages that have been erased many times are assigned "cold data," that is, data that are rarely updated, while pages that have not been erased many times are used to store "hot data," that is, data that are updated frequently. This principle of evenly distributing erase operations across physical blocks is called **wear leveling**, and is usually performed transparently by flash-memory controllers. If a physical page is damaged due to an excessive number of updates, it can be removed from usage, without affecting the flash memory as a whole.

All the above actions are carried out by a layer of software called the **flash translation layer**; above this layer, flash storage looks identical to magnetic disk storage, providing the same page/sector-oriented interface, except that flash storage is much faster. File systems and database storage structures can thus see an identical logical view of the underlying storage structure, regardless of whether it is flash or magnetic storage.

**Hybrid disk drives** are hard-disk systems that combine magnetic storage with a smaller amount of flash memory, which is used as a cache for frequently

accessed data. Frequently accessed data that are rarely updated are ideal for caching in flash memory.

## 10.3 RAID

The data-storage requirements of some applications (in particular Web, database, and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk-drive capacities have been growing very fast.

Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Several independent reads or writes can also be performed in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data.

A variety of disk-organization techniques, collectively called **redundant arrays of independent disks** (**RAID**), have been proposed to achieve improved performance and reliability.

In the past, system designers viewed storage systems composed of several small, cheap disks as a cost-effective alternative to using large, expensive disks; the cost per megabyte of the smaller disks was less than that of larger disks. In fact, the I in RAID, which now stands for *independent*, originally stood for *inexpensive*. Today, however, all disks are physically small, and larger-capacity disks actually have a lower cost per megabyte. RAID systems are used for their higher reliability and higher performance rate, rather than for economic reasons. Another key justification for RAID use is easier management and operations.

### 10.3.1 Improvement of Reliability via Redundancy

Let us first consider reliability. The chance that at least one disk out of a set of $N$ disks will fail is much higher than the chance that a specific single disk will fail. Suppose that the mean time to failure of a disk is 100,000 hours, or slightly over 11 years. Then, the mean time to failure of some disk in an array of 100 disks will be 100,000/100 = 1000 hours, or around 42 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data (as discussed in Section 10.2.1). Such a high frequency of data loss is unacceptable.

The solution to the problem of reliability is to introduce **redundancy**; that is, we store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost, so the effective mean time to failure is increased, provided that we count only failures that lead to loss of data or to nonavailability of data.

The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring** (or, sometimes, *shadowing*). A logical disk then consists of two physical disks, and every write is carried

out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is repaired.

The mean time to failure (where failure is the loss of data) of a mirrored disk depends on the mean time to failure of the individual disks, as well as on the **mean time to repair**, which is the time it takes (on an average) to replace a failed disk and to restore the data on it. Suppose that the failures of the two disks are *independent*; that is, there is no connection between the failure of one disk and the failure of the other. Then, if the mean time to failure of a single disk is 100,000 hours, and the mean time to repair is 10 hours, the **mean time to data loss** of a mirrored disk system is $100,000^2/(2 * 10) = 500 * 10^6$ hours, or 57,000 years! (We do not go into the derivations here; references in the bibliographical notes provide the details.)

You should be aware that the assumption of independence of disk failures is not valid. Power failures, and natural disasters such as earthquakes, fires, and floods, may result in damage to both disks at the same time. As disks age, the probability of failure increases, increasing the chance that a second disk will fail while the first is being repaired. In spite of all these considerations, however, mirrored-disk systems offer much higher reliability than do single-disk systems. Mirrored-disk systems with mean time to data loss of about 500,000 to 1,000,000 hours, or 55 to 110 years, are available today.

Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. Power failures are not a concern if there is no data transfer to disk in progress when they occur. However, even with mirroring of disks, if writes are in progress to the same block in both disks, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state. The solution to this problem is to write one copy first, then the next, so that one of the two copies is always consistent. Some extra actions are required when we restart after a power failure, to recover from incomplete writes. This matter is examined in Practice Exercise 10.3.

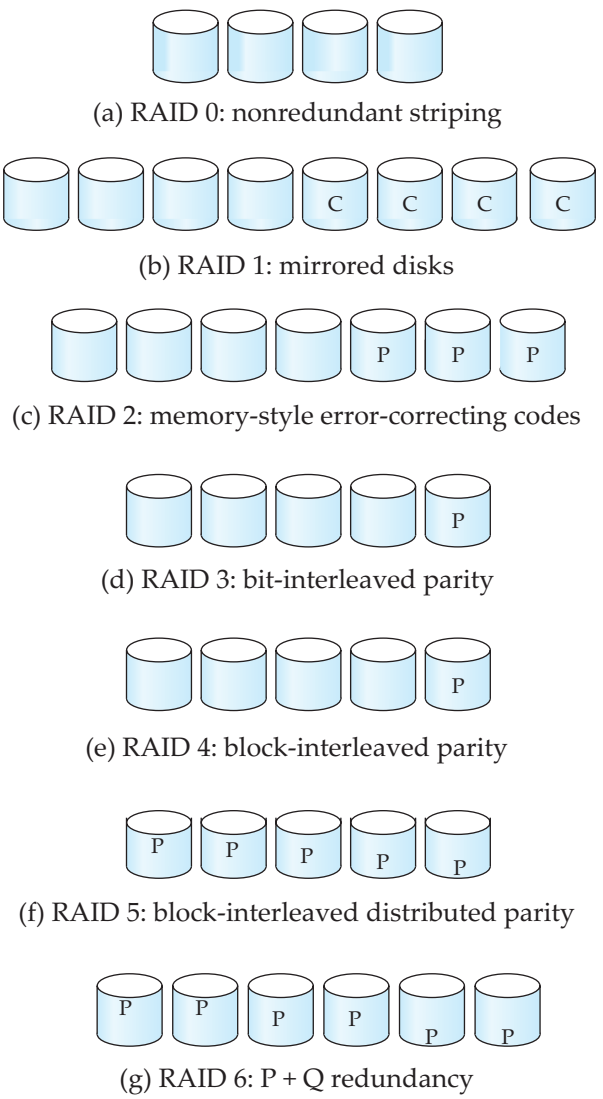
### 10.3.2 Improvement in Performance via Parallelism

Now let us consider the benefit of parallel access to multiple disks. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

With multiple disks, we can improve the transfer rate as well (or instead) by **striping data** across multiple disks. In its simplest form, data striping consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**. For example, if we have an array of eight disks, we write bit $i$ of each byte to disk $i$. The array of eight disks can be treated as a single disk with sectors that are eight times the normal size, and, more important, that has eight times the transfer rate. In such an organization, every disk participates in every access (read or write), so the number of accesses that can be processed per second is about the same as on a single disk, but each access can read eight times as many

data in the same time as on a single disk. Bit-level striping can be generalized to a number of disks that either is a multiple of 8 or a factor of 8. For example, if we use an array of four disks, bits $i$ and $4 + i$ of each byte go to disk $i$.

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




(a) RAID 0: nonredundant striping

(b) RAID 1: mirrored disks

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

(d) RAID 3: bit-interleaved parity

(e) RAID 4: block-interleaved parity

(f) RAID 5: block-interleaved distributed parity

(g) RAID 6: P + Q redundancy

**Figure 10.3** RAID levels.

block of the disk to store logical block $i$. For example, with 8 disks, logical block 0 is stored in physical block 0 of disk 1, while logical block 11 is stored in physical block 1 of disk 4. When reading a large file, block-level striping fetches $n$ blocks at a time in parallel from the $n$ disks, giving a high data-transfer rate for large reads. When a single block is read, the data-transfer rate is the same as on one disk, but the remaining $n - 1$ disks are free to perform other actions.

Block-level striping is the most commonly used form of data striping. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible.

In summary, there are two main goals of parallelism in a disk system:

1. Load-balance multiple small accesses (block accesses), so that the throughput of such accesses increases.
2. Parallelize large accesses so that the response time of large accesses is reduced.

### 10.3.3 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but does not improve reliability. Various alternative schemes aim to provide redundancy at lower cost by combining disk striping with "parity" bits (which we describe next). These schemes have different cost–performance trade-offs. The schemes are classified into **RAID levels**, as in Figure 10.3. (In the figure, P indicates error-correcting bits, and C indicates a second copy of the data.) For all levels, the figure depicts four disks' worth of data, and the extra disks depicted are used to store redundant information for failure recovery.

- **RAID level 0** refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits). Figure 10.3a shows an array of size 4.
- **RAID level 1** refers to disk mirroring with block striping. Figure 10.3b shows a mirrored organization that holds four disks' worth of data.
  Note that some vendors use the term **RAID level 1+0** or **RAID level 10** to refer to mirroring with striping, and use the term RAID level 1 to refer to mirroring without striping. Mirroring without striping can also be used with arrays of disks, to give the appearance of a single large, reliable disk: if each disk has $M$ blocks, logical blocks 0 to $M - 1$ are stored on disk 0, $M$ to $2M - 1$ on disk 1(the second disk), and so on, and each disk is mirrored.[2]
- **RAID level 2**, known as memory-style error-correcting-code (ECC) organization, employs parity bits. Memory systems have long used parity bits for

[2]Note that some vendors use the term RAID 0+1 to refer to a version of RAID that uses striping to create a RAID 0 array, and mirrors the array onto another array, with the difference from RAID 1 being that if a disk fails, the RAID 0 array containing the disk becomes unusable. The mirrored array can still be used, so there is no loss of data. This arrangement is inferior to RAID 1 when a disk has failed, since the other disks in the RAID 0 array can continue to be used in RAID 1, but remain idle in RAID 0+1.

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

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

Figure 10.3c shows the level 2 scheme. The disks labeled *P* store the error-correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks, and can be used to reconstruct the damaged data. Figure 10.3c shows an array of size 4; note RAID level 2 requires only three disks' overhead for four disks of data, unlike RAID level 1, which required four disks' overhead.

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

  RAID level 3 is as good as level 2, but is less expensive in the number of extra disks (it has only a one-disk overhead), so level 2 is not used in practice. Figure 10.3d shows the level 3 scheme.

  RAID level 3 has two benefits over level 1. It needs only one parity disk for several regular disks, whereas level 1 needs one mirror disk for every disk, and thus level 3 reduces the storage overhead. Since reads and writes of a byte are spread out over multiple disks, with *N*-way striping of data, the transfer rate for reading or writing a single block is *N* times faster than a RAID level 1 organization using *N*-way striping. On the other hand, RAID level 3 supports a lower number of I/O operations per second, since every disk has to participate in every I/O request.

- **RAID level 4**, block-interleaved parity organization, uses block-level striping, like RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from *N* other disks. This scheme is shown pictorially in Figure 10.3e. If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

Small independent writes, on the other hand, cannot be performed in parallel. A write of a block has to access the disk on which the block is stored, as well as the parity disk, since the parity block has to be updated. Moreover, both the old value of the parity block and the old value of the block being written have to be read for the new parity to be computed. Thus, a single write requires four disk accesses: two to read the two old blocks, and two to write the two blocks.

- **RAID level 5**, block-interleaved distributed parity, improves on level 4 by partitioning data and parity among all $N + 1$ disks, instead of storing data in $N$ disks and parity in one disk. In level 5, all disks can participate in satisfying read requests, unlike RAID level 4, where the parity disk cannot participate, so level 5 increases the total number of requests that can be met in a given amount of time. For each set of $N$ logical blocks, one of the disks stores the parity, and the other $N$ disks store the blocks.

  Figure 10.3f shows the setup. The P's are distributed across all the disks. For example, with an array of 5 disks, the parity block, labeled Pk, for logical blocks $4k, 4k + 1, 4k + 2, 4k + 3$ is stored in disk $k$ mod 5; the corresponding blocks of the other four disks store the 4 data blocks $4k$ to $4k+3$. The following table indicates how the first 20 blocks, numbered 0 to 19, and their parity blocks are laid out. The pattern shown gets repeated on further blocks.

| | | | | |
|---|---|---|---|---|
| P0 | 0 | 1 | 2 | 3 |
| 4 | P1 | 5 | 6 | 7 |
| 8 | 9 | P2 | 10 | 11 |
| 12 | 13 | 14 | P3 | 15 |
| 16 | 17 | 18 | 19 | P4 |

  Note that a parity block cannot store parity for blocks in the same disk, since then a disk failure would result in loss of data as well as of parity, and hence would not be recoverable. Level 5 subsumes level 4, since it offers better read–write performance at the same cost, so level 4 is not used in practice.

- **RAID level 6**, the P + Q redundancy scheme, is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures. Instead of using parity, level 6 uses error-correcting codes such as the Reed–Solomon codes (see the bibliographical notes). In the scheme in Figure 10.3g, 2 bits of redundant data are stored for every 4 bits of data—unlike 1 parity bit in level 5—and the system can tolerate two disk failures.

Finally, we note that several variations have been proposed to the basic RAID schemes described here, and different vendors use different terminologies for the variants.

### 10.3.4 Choice of RAID Level

The factors to be taken into account in choosing a RAID level are:

- Monetary cost of extra disk-storage requirements.
- Performance requirements in terms of number of I/O operations.
- Performance when a disk has failed.
- Performance during rebuild (that is, while the data in a failed disk are being rebuilt on a new disk).

The time to rebuild the data of a failed disk can be significant, and it varies with the RAID level that is used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data of a failed disk. The **rebuild performance** of a RAID system may be an important factor if continuous availability of data is required, as it is in high-performance database systems. Furthermore, since rebuild time can form a significant part of the repair time, rebuild performance also influences the mean time to data loss.

RAID level 0 is used in high-performance applications where data safety is not critical. Since RAID levels 2 and 4 are subsumed by RAID levels 3 and 5, the choice of RAID levels is restricted to the remaining levels. Bit striping (level 3) is inferior to block striping (level 5), since block striping gives as good data-transfer rates for large transfers, while using fewer disks for small transfers. For small transfers, the disk access time dominates anyway, so the benefit of parallel reads diminishes. In fact, level 3 may perform worse than level 5 for a small transfer, since the transfer completes only when corresponding sectors on all disks have been fetched; the average latency for the disk array thus becomes very close to the worst-case latency for a single disk, negating the benefits of higher transfer rates. Level 6 is not supported currently by many RAID implementations, but it offers better reliability than level 5 and can be used in applications where data safety is very important.

The choice between RAID level 1 and level 5 is harder to make. RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance. RAID level 5 has a lower storage overhead than level 1, but has a higher time overhead for writes. For applications where data are read frequently, and written rarely, level 5 is the preferred choice.

Disk-storage capacities have been growing at a rate of over 50 percent per year for many years, and the cost per byte has been falling at the same rate. As a result, for many existing database applications with moderate storage requirements, the monetary cost of the extra disk storage needed for mirroring has become relatively small (the extra monetary cost, however, remains a significant issue for storage-

intensive applications such as video data storage). Access speeds have improved at a much slower rate (around a factor of 3 over 10 years), while the number of I/O operations required per second has increased tremendously, particularly for Web application servers.

RAID level 5, which increases the number of I/O operations needed to write a single logical block, pays a significant time penalty in terms of write performance. RAID level 1 is therefore the RAID level of choice for many applications with moderate storage requirements and high I/O requirements.

RAID system designers have to make several other decisions as well. For example, how many disks should there be in an array? How many bits should be protected by each parity bit? If there are more disks in an array, data-transfer rates are higher, but the system will be more expensive. If there are more bits protected by a parity bit, the space overhead due to parity bits is lower, but there is an increased chance that a second disk will fail before the first failed disk is repaired, and that will result in data loss.

### 10.3.5 Hardware Issues

Another issue in the choice of RAID implementations is at the level of hardware. RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called **software RAID**. However, there are significant benefits to be had by building special-purpose hardware to support RAID, which we outline below; systems with special hardware support are called **hardware RAID** systems.

Hardware RAID implementations can use nonvolatile RAM to record writes before they are performed. In case of power failure, when the system comes back up, it retrieves information about any incomplete writes from nonvolatile RAM and then completes the writes. Without such hardware support, extra work needs to be done to detect blocks that may have been partially written before power failure (see Practice Exercise 10.3).

Even if all writes are completed properly, there is a small chance of a sector in a disk becoming unreadable at some point, even though it was successfully written earlier. Reasons for loss of data on individual sectors could range from manufacturing defects, to data corruption on a track when an adjacent track is written repeatedly. Such loss of data that were successfully written earlier is sometimes referred to as a *latent failure*, or as *bit rot*. When such a failure happens, if it is detected early the data can be recovered from the remaining disks in the RAID organization. However, if such a failure remains undetected, a single disk failure could lead to data loss if a sector in one of the other disks has a latent failure.

To minimize the chance of such data loss, good RAID controllers perform **scrubbing**; that is, during periods when disks are idle, every sector of every disk is read, and if any sector is found to be unreadable, the data are recovered from the remaining disks in the RAID organization, and the sector is written back. (If the physical sector is damaged, the disk controller would remap the logical sector address to a different physical sector on disk.)

Some hardware RAID implementations permit **hot swapping**; that is, faulty disks can be removed and replaced by new ones without turning power off. Hot swapping reduces the mean time to repair, since replacement of a disk does not have to wait until a time when the system can be shut down. In fact many critical systems today run on a 24 × 7 schedule; that is, they run 24 hours a day, 7 days a week, providing no time for shutting down and replacing a failed disk. Further, many RAID implementations assign a spare disk for each array (or for a set of disk arrays). If a disk fails, the spare disk is immediately used as a replacement. As a result, the mean time to repair is reduced greatly, minimizing the chance of any data loss. The failed disk can be replaced at leisure.

The power supply, or the disk controller, or even the system interconnection in a RAID system could become a single point of failure that could stop functioning of the RAID system. To avoid this possibility, good RAID implementations have multiple redundant power supplies (with battery backups so they continue to function even if power fails). Such RAID systems have multiple disk interfaces, and multiple interconnections to connect the RAID system to the computer system (or to a network of computer systems). Thus, failure of any single component will not stop the functioning of the RAID system.

### 10.3.6 Other RAID Applications

The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, the RAID structures are able to recover data even if one of the tapes in an array of tapes is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit; if one of the units is not received for any reason, it can be reconstructed from the other units.

## 10.4 Tertiary Storage

In a large database system, some of the data may have to reside on tertiary storage. The two most common tertiary storage media are optical disks and magnetic tapes.

### 10.4.1 Optical Disks

Compact disks have been a popular medium for distributing software, multimedia data such as audio and images, and other electronically published information. They have a storage capacity of 640 to 700 megabytes, and they are cheap to mass-produce. Digital video disks (DVDs) have now replaced compact disks in applications that require larger amounts of data. Disks in the DVD-5 format can store 4.7 gigabytes of data (in one recording layer), while disks in the DVD-9 format can store 8.5 gigabytes of data (in two recording layers). Recording on both sides of a disk yields even larger capacities; DVD-10 and DVD-18 formats, which are the two-sided versions of DVD-5 and DVD-9, can store 9.4 gigabytes

and 17 gigabytes, respectively. The *Blu-ray DVD* format has a significantly higher capacity of 27 to 54 gigabytes per disk.

CD and DVD drives have much longer seek times (100 milliseconds is common) than do magnetic-disk drives, since the head assembly is heavier. Rotational speeds are typically lower than those of magnetic disks, although the faster CD and DVD drives have rotation speeds of about 3000 rotations per minute, which is comparable to speeds of lower-end magnetic-disk drives. Rotational speeds of CD drives originally corresponded to the audio CD standards, and the speeds of DVD drives originally corresponded to the DVD video standards, but current-generation drives rotate at many times the standard rate.

Data-transfer rates are somewhat less than for magnetic disks. Current CD drives read at around 3 to 6 megabytes per second, and current DVD drives read at 8 to 20 megabytes per second. Like magnetic-disk drives, optical disks store more data in outside tracks and less data in inner tracks. The transfer rate of optical drives is characterized as $n\times$, which means the drive supports transfers at $n$ times the standard rate; rates of around $50\times$ for CD and $16\times$ for DVD are now common.

The record-once versions of optical disks (CD-R, DVD-R, and DVD+R) are popular for distribution of data and particularly for archival storage of data because they have a high capacity, have a longer lifetime than magnetic disks, and can be removed and stored at a remote location. Since they cannot be overwritten, they can be used to store information that should not be modified, such as audit trails. The multiple-write versions (CD-RW, DVD-RW, DVD+RW, and DVD-RAM) are also used for archival purposes.

**Jukeboxes** are devices that store a large number of optical disks (up to several hundred) and load them automatically on demand to one of a small number of drives (usually 1 to 10). The aggregate storage capacity of such a system can be many terabytes. When a disk is accessed, it is loaded by a mechanical arm from a rack onto a drive (any disk that was already in the drive must first be placed back on the rack). The disk load/unload time is usually of the order of a few seconds —very much longer than disk access times.

### 10.4.2 Magnetic Tapes

Although magnetic tapes are relatively permanent, and can hold large volumes of data, they are slow in comparison to magnetic and optical disks. Even more important, magnetic tapes are limited to sequential access. Thus, they cannot provide random access for secondary-storage requirements, although historically, prior to the use of magnetic disks, tapes were used as a secondary-storage medium.

Tapes are used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another. Tapes are also used for storing large volumes of data, such as video or image data, that either do not need to be accessible quickly or are so voluminous that magnetic-disk storage would be too expensive.

A tape is kept in a spool, and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take seconds or even minutes, rather than

milliseconds; once positioned, however, tape drives can write data at densities and speeds approaching those of disk drives. Capacities vary, depending on the length and width of the tape and on the density at which the head can read and write. The market is currently fragmented among a wide variety of tape formats. Currently available tape capacities range from a few gigabytes with the **Digital Audio Tape** (DAT) format, 10 to 40 gigabytes with the **Digital Linear Tape** (DLT) format, 100 gigabytes and higher with the **Ultrium** format, to 330 gigabytes with **Ampex helical scan** tape formats. Data-transfer rates are of the order of a few to tens of megabytes per second.

Tape devices are quite reliable, and good tape drive systems perform a read of the just-written data to ensure that it has been recorded correctly. Tapes, however, have limits on the number of times that they can be read or written reliably.

**Tape jukeboxes**, like optical disk jukeboxes, hold large numbers of tapes, with a few drives onto which the tapes can be mounted; they are used for storing large volumes of data, ranging up to many petabytes ($10^{15}$ bytes), with access times on the order of seconds to a few minutes. Applications that need such enormous data storage include imaging systems that gather data from remote-sensing satellites, and large video libraries for television broadcasters.

Some tape formats (such as the Accelis format) support faster seek times (of the order of tens of seconds), and are intended for applications that retrieve information from jukeboxes. Most other tape formats provide larger capacities, at the cost of slower access; such formats are ideal for data backup, where fast seeks are not important.

Tape drives have been unable to keep up with the enormous improvements in disk drive capacity and corresponding reduction in storage cost. While the cost of tapes is low, the cost of tape drives and tape libraries is significantly higher than the cost of a disk drive: a tape library capable of storing a few terabytes can costs tens of thousands of dollars. Backing up data to disk drives has become a cost-effective alternative to tape backup for a number of applications.

## 10.5 File Organization

A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks. A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying *file system*. We need to consider ways of representing logical data models in terms of files.

Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer. Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created. Larger block sizes can be useful in some database applications.

A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used. We

shall assume that *no record is larger than a block*. This assumption is realistic for most data-processing applications, such as our university example. There are certainly several kinds of large data items, such as images, that can be significantly larger than a block. We briefly discuss how to handle such large data items later, in Section 10.5.2, by storing large data items separately, and storing a pointer to the data item in the record.

In addition, we shall require that *each record is entirely contained in a single block*; that is, no record is contained partly in one block, and partly in another. This restriction simplifies and speeds up access to data items.

In a relational database, tuples of distinct relations are generally of different sizes. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records; however, files of fixed-length records are easier to implement than are files of variable-length records. Many of the techniques used for the former can be applied to the variable-length case. Thus, we begin by considering a file of fixed-length records, and consider storage of variable-length records later.

### 10.5.1 Fixed-Length Records

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

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

Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes. Suppose that instead of allocating a variable amount of bytes for the attributes *ID*, *name*, and *dept_name*, we allocate the maximum number of bytes that each attribute can hold. Then, the *instructor* record is 53 bytes long. A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on (Figure 10.4). However, there are two problems with this simple approach:

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

| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

**Figure 10.4** File containing *instructor* records.

To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.

When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead (Figure 10.5). Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record (Figure 10.6).

It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the

| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

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

| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |

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

deleted record, and to wait for a subsequent insertion before reusing the space. A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as *pointers*, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a **free list**. Figure 10.7 shows the file of Figure 10.4, with the free list, after records 1, 4, and 6 have been deleted.

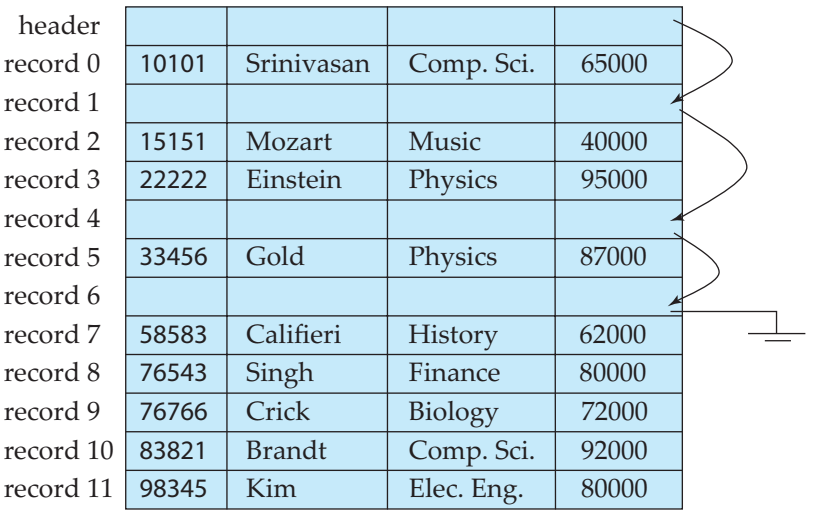
On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.

Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record. If we allow records of variable length in a file, this match no longer holds. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.

### 10.5.2 Variable-Length Records

Variable-length records arise in database systems in several ways:

- Storage of multiple record types in a file.
- Record types that allow variable lengths for one or more fields.
- Record types that allow repeating fields, such as arrays or multisets.



| | | | | |
|---|---|---|---|---|
| header | | | | |
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | | | | |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | | | | |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | | | | |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

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

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

- How to represent a single record in such a way that individual attributes can be extracted easily.
- How to store variable-length records within a block, such that records in a block can be extracted easily.

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

An example of such a record representation is shown in Figure 10.8. The figure shows an *instructor* record, whose first three attributes *ID*, *name*, and *dept_name* are variable-length strings, and whose fourth attribute *salary* is a fixed-sized number. We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The *salary* attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.

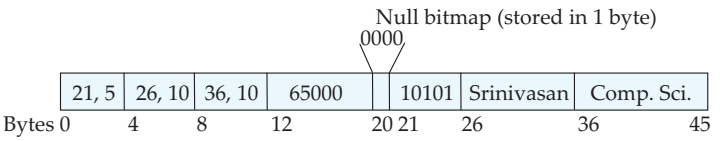



**Figure 10.8** Representation of variable-length record.

The figure also illustrates the use of a **null bitmap**, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the *salary* value stored in bytes 12 through 19 would be ignored. Since the record has four attributes, the null bitmap for this record fits in 1 byte, although more bytes may be required with more attributes. In some representations, the null bitmap is stored at the beginning of the record, and for attributes that are null, no data (value, or offset/length) are stored at all. Such a representation would save some storage space, at the cost of extra work to extract attributes of the record. This representation is particularly useful for certain applications where records have a large number of fields, most of which are null.

We next address the problem of storing variable-length records in a block. The **slotted-page structure** is commonly used for organizing records within a block, and is shown in Figure 10.9.[3] There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header.
2. The end of free space in the block.
3. An array whose entries contain the location and size of each record.

The actual records are allocated *contiguously* in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

If a record is deleted, the space that it occupies is freed, and its entry is set to *deleted* (its size is set to −1, for example). Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the header array and the first record. The end-of-free-space pointer in the header is appropriately updated as well. Records can be grown or shrunk by similar techniques, as long as there is space in the block. The cost of moving the records is not too high, since the size of a block is limited: typical values are around 4 to 8 kilobytes.

The slotted-page structure requires that there be no pointers that point directly to records. Instead, pointers must point to the entry in the header that contains the

[3]Here, "page" is synonymous with "block."

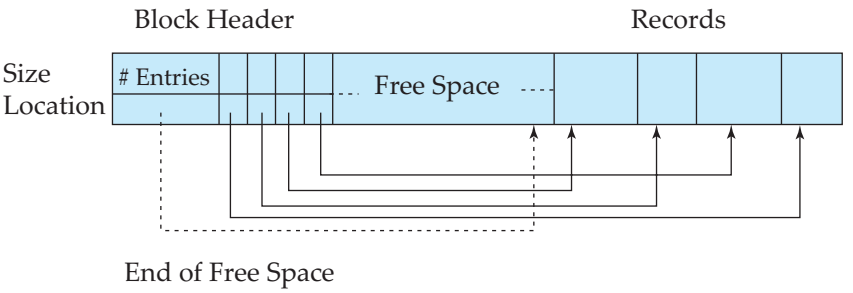



**Figure 10.9** Slotted-page structure.

actual location of the record. This level of indirection allows records to be moved to prevent fragmentation of space inside a block, while supporting indirect pointers to the record.

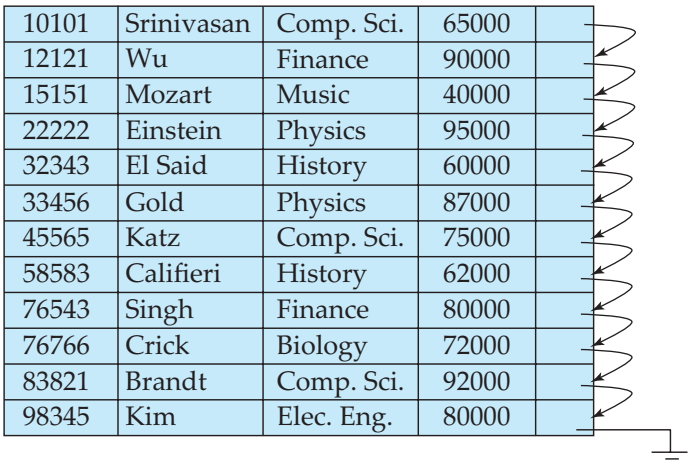
Databases often store data that can be much larger than a disk block. For instance, an image or an audio recording may be multiple megabytes in size, while a video object may be multiple gigabytes in size. Recall that SQL supports the types **blob** and **clob**, which store binary and character large objects.

Most relational databases restrict the size of a record to be no larger than the size of a block, to simplify buffer management and free-space management. Large objects are often stored in a special file (or collection of files) instead of being stored with the other (short) attributes of records in which they occur. A (logical) pointer to the object is then stored in the record containing the large object. Large objects are often represented using $B^+$-tree file organizations, which we study in Section 11.4.1. $B^+$-tree file organizations permit us to read an entire object, or specified byte ranges in the object, as well as to insert and delete parts of the object.

## 10.6 Organization of Records in Files

So far, we have studied how records are represented in a file structure. A relation is a set of records. Given a set of records, the next question is how to organize them in a file. Several of the possible ways of organizing records in files are:

- **Heap file organization**. Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
- **Sequential file organization**. Records are stored in sequential order, according to the value of a "search key" of each record. Section 10.6.1 describes this organization.
- **Hashing file organization**. A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the



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

**Figure 10.10** Sequential file for *instructor* records.

file the record should be placed. Chapter 11 describes this organization; it is closely related to the indexing structures described in that chapter.

Generally, a separate file is used to store the records of each relation. However, in a **multitable clustering file organization**, records of several different relations are stored in the same file; further, related records of the different relations are stored on the same block, so that one I/O operation fetches related records from all the relations. For example, records of the two relations can be considered to be related if they would match in a join of the two relations. Section 10.6.2 describes this organization.

### 10.6.1 Sequential File Organization

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

Figure 10.10 shows a sequential file of *instructor* records taken from our university example. In that example, the records are stored in search-key order, using *ID* as the search key.

The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms that we shall study in Chapter 12.

It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single

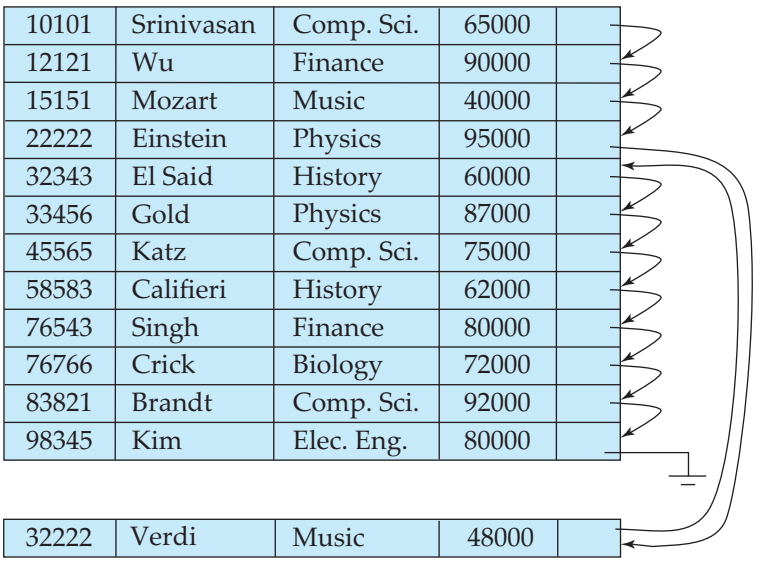



**Figure 10.11** Sequential file after an insertion.

insertion or deletion. We can manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following rules:

1. Locate the record in the file that comes before the record to be inserted in search-key order.
2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an *overflow block*. In either case, adjust the pointers so as to chain together the records in search-key order.

Figure 10.11 shows the file of Figure 10.10 after the insertion of the record (32222, Verdi, Music, 48000). The structure in Figure 10.11 allows fast insertion of new records, but forces sequential file-processing applications to process records in an order that does not match the physical order of the records.

If relatively few records need to be stored in overflow blocks, this approach works well. Eventually, however, the correspondence between search-key order and physical order may be totally lost over a period of time, in which case sequential processing will become much less efficient. At this point, the file should be **reorganized** so that it is once again physically in sequential order. Such reorganizations are costly, and must be done during times when the system load is low. The frequency with which reorganizations are needed depends on the frequency of insertion of new records. In the extreme case in which insertions rarely occur, it is possible always to keep the file in physically sorted order. In such a case, the pointer field in Figure 10.10 is not needed.

### 10.6.2 Multitable Clustering File Organization

Many relational database systems store each relation in a separate file, so that they can take full advantage of the file system that the operating system provides. Usually, tuples of a relation can be represented as fixed-length records. Thus, relations can be mapped to a simple file structure. This simple implementation of a relational database system is well suited to low-cost database implementations as in, for example, embedded systems or portable devices. In such systems, the size of the database is small, so little is gained from a sophisticated file structure. Furthermore, in such environments, it is essential that the overall size of the object code for the database system be small. A simple file structure reduces the amount of code needed to implement the system.

This simple approach to relational database implementation becomes less satisfactory as the size of the database increases. We have seen that there are performance advantages to be gained from careful assignment of records to blocks, and from careful organization of the blocks themselves. Clearly, a more complicated file structure may be beneficial, even if we retain the strategy of storing each relation in a separate file.

However, many large-scale database systems do not rely directly on the underlying operating system for file management. Instead, one large operating-system file is allocated to the database system. The database system stores all relations in this one file, and manages the file itself.

Even if multiple relations are stored in a single file, by default most databases store records of only one relation in a given block. This simplifies data management. However, in some cases it can be useful to store records of more than one relation in a single block. To see the advantage of storing records of multiple relations in one block, consider the following SQL query for the university database:

```
select dept_name, building, budget, ID, name, salary
from department natural join instructor;
```

This query computes a join of the *department* and *instructor* relations. Thus, for each tuple of *department*, the system must locate the *instructor* tuples with the same value for *dept_name*. Ideally, these records will be located with the help of *indices*, which we shall discuss in Chapter 11. Regardless of how these records are located, however, they need to be transferred from disk into main memory. In the worst case, each record will reside on a different block, forcing us to do one block read for each record required by the query.

| *dept_name* | *building* | *budget* |
|---|---|---|
| Comp. Sci. | Taylor | 100000 |
| Physics | Watson | 70000 |

**Figure 10.12** The *department* relation.

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |

**Figure 10.13** The *instructor* relation.

As a concrete example, consider the *department* and *instructor* relations of Figures 10.12 and 10.13, respectively (for brevity, we include only a subset of the tuples of the relations we have used thus far). In Figure 10.14, we show a file structure designed for efficient execution of queries involving the natural join of *department* and *instructor*. The *instructor* tuples for each *ID* are stored near the *department* tuple for the corresponding *dept_name*. This structure mixes together tuples of two relations, but allows for efficient processing of the join. When a tuple of the *department* relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding *instructor* tuples are stored on the disk near the *department* tuple, the block containing the *department* tuple contains tuples of the *instructor* relation needed to process the query. If a department has so many instructors that the *instructor* records do not fit in one block, the remaining records appear on nearby blocks.

A **multitable clustering file organization** is a file organization, such as that illustrated in Figure 10.14, that stores related records of two or more relations in each block. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently.

In the representation shown in Figure 10.14, the *dept_name* attribute is omitted from *instructor* records since it can be inferred from the associated *department* record; the attribute may be retained in some implementations, to simplify access to the attributes. We assume that each record contains the identifier of the relation to which it belongs, although this is not shown in Figure 10.14.

Our use of clustering of multiple tables into a single file has enhanced processing of a particular join (that of *department* and *instructor*), but it results in slowing processing of other types of queries. For example,

| | | |
|---|---|---|
| Comp. Sci. | Taylor | 100000 |
| 45564 | Katz | 75000 |
| 10101 | Srinivasan | 65000 |
| 83821 | Brandt | 92000 |
| Physics | Watson | 70000 |
| 33456 | Gold | 87000 |

**Figure 10.14** Multitable clustering file structure.

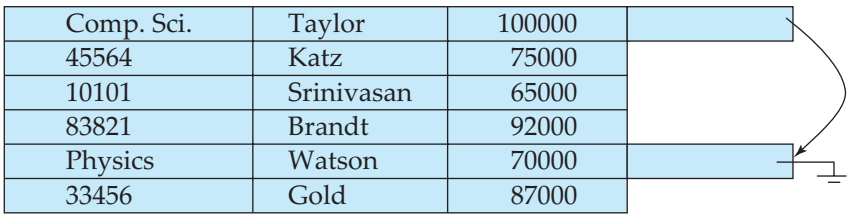



**Figure 10.15** Multitable clustering file structure with pointer chains.

**select** *
**from** *department*;

requires more block accesses than it did in the scheme under which we stored each relation in a separate file, since each block now contains significantly fewer *department* records. To locate efficiently all tuples of the *department* relation in the structure of Figure 10.14, we can chain together all the records of that relation using pointers, as in Figure 10.15.

When multitable clustering is to be used depends on the types of queries that the database designer believes to be most frequent. Careful use of multitable clustering can produce significant performance gains in query processing.

## 10.7 Data-Dictionary Storage

So far, we have considered only the representation of the relations themselves. A relational database system needs to maintain data *about* the relations, such as the schema of the relations. In general, such "data about data" is referred to as **metadata**.

Relational schemas and other metadata about relations are stored in a structure called the **data dictionary** or **system catalog**. Among the types of information that the system must store are these:

- Names of the relations.
- Names of the attributes of each relation.
- Domains and lengths of attributes.
- Names of views defined on the database, and definitions of those views.
- Integrity constraints (for example, key constraints).

In addition, many systems keep the following data on users of the system:

- Names of authorized users.
- Authorization and accounting information about users.

- Passwords or other information used to authenticate users.

Further, the database may store statistical and descriptive data about the relations, such as:

- Number of tuples in each relation.
- Method of storage for each relation (for example, clustered or nonclustered).

The data dictionary may also note the storage organization (sequential, hash, or heap) of relations, and the location where each relation is stored:

- If relations are stored in operating system files, the dictionary would note the names of the file (or files) containing each relation.
- If the database stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list.

In Chapter 11, in which we study indices, we shall see a need to store information about each index on each of the relations:

- Name of the index.
- Name of the relation being indexed.
- Attributes on which the index is defined.
- Type of index formed.

All this metadata information constitutes, in effect, a miniature database. Some database systems store such metadata by using special-purpose data structures and code. It is generally preferable to store the data about the database as relations in the database itself. By using database relations to store system metadata, we simplify the overall structure of the system and harness the full power of the database for fast access to system data.

The exact choice of how to represent system metadata by relations must be made by the system designers. One possible representation, with primary keys underlined, is shown in Figure 10.16. In this representation, the attribute *index_attributes* of the relation *Index_metadata* is assumed to contain a list of one or more attributes, which can be represented by a character string such as "*dept_name*, *building*". The *Index_metadata* relation is thus not in first normal form; it can be normalized, but the above representation is likely to be more efficient to access. The data dictionary is often stored in a nonnormalized form to achieve fast access.

Whenever the database system needs to retrieve records from a relation, it must first consult the *Relation_metadata* relation to find the location and storage organization of the relation, and then fetch records using this information. However, the storage organization and location of the *Relation_metadata* relation itself

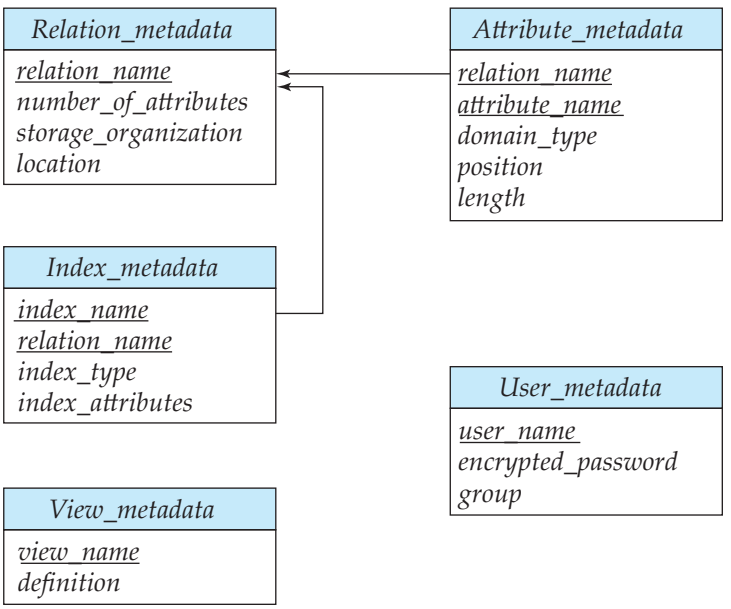



**Figure 10.16** Relational schema representing system metadata.

must be recorded elsewhere (for example, in the database code itself, or in a fixed location in the database), since we need this information to find the contents of *Relation_metadata*.

## 10.8 Database Buffer

A major goal of the database system is to minimize the number of block transfers between the disk and memory. One way to reduce the number of disk accesses is to keep as many blocks as possible in main memory. The goal is to maximize the chance that, when a block is accessed, it is already in main memory, and, thus, no disk access is required.

Since it is not possible to keep all blocks in main memory, we need to manage the allocation of the space available in main memory for the storage of blocks. The **buffer** is that part of main memory available for storage of copies of disk blocks. There is always a copy kept on disk of every block, but the copy on disk may be a version of the block older than the version in the buffer. The subsystem responsible for the allocation of buffer space is called the **buffer manager**.

### 10.8.1 Buffer Manager

Programs in a database system make requests (that is, calls) on the buffer manager when they need a block from disk. If the block is already in the buffer, the buffer manager passes the address of the block in main memory to the requester. If the

block is not in the buffer, the buffer manager first allocates space in the buffer for the block, throwing out some other block, if necessary, to make space for the new block. The thrown-out block is written back to disk only if it has been modified since the most recent time that it was written to the disk. Then, the buffer manager reads in the requested block from the disk to the buffer, and passes the address of the block in main memory to the requester. The internal actions of the buffer manager are transparent to the programs that issue disk-block requests.

If you are familiar with operating-system concepts, you will note that the buffer manager appears to be nothing more than a virtual-memory manager, like those found in most operating systems. One difference is that the size of the database might be larger than the hardware address space of a machine, so memory addresses are not sufficient to address all disk blocks. Further, to serve the database system well, the buffer manager must use techniques more sophisticated than typical virtual-memory management schemes:

- **Buffer replacement strategy**. When there is no room left in the buffer, a block must be removed from the buffer before a new one can be read in. Most operating systems use a **least recently used** (**LRU**) scheme, in which the block that was referenced least recently is written back to disk and is removed from the buffer. This simple approach can be improved on for database applications.
- **Pinned blocks**. For the database system to be able to recover from crashes (Chapter 16), it is necessary to restrict those times when a block may be written back to disk. For instance, most recovery systems require that a block should not be written to disk while an update on the block is in progress. A block that is not allowed to be written back to disk is said to be **pinned**. Although many operating systems do not support pinned blocks, such a feature is essential for a database system that is resilient to crashes.
- **Forced output of blocks**. There are situations in which it is necessary to write back the block to disk, even though the buffer space that it occupies is not needed. This write is called the **forced output** of a block. We shall see the reason for forced output in Chapter 16; briefly, main-memory contents and thus buffer contents are lost in a crash, whereas data on disk usually survive a crash.

### 10.8.2 Buffer-Replacement Policies

The goal of a replacement strategy for blocks in the buffer is to minimize accesses to the disk. For general-purpose programs, it is not possible to predict accurately which blocks will be referenced. Therefore, operating systems use the past pattern of block references as a predictor of future references. The assumption generally made is that blocks that have been referenced recently are likely to be referenced again. Therefore, if a block must be replaced, the least recently referenced block is replaced. This approach is called the **least recently used** (**LRU**) block-replacement scheme.

```
for each tuple i of instructor do
    for each tuple d of department do
        if i[dept_name] = d[dept_name]
        then begin
                let x be a tuple defined as follows:
                x[ID] := i[ID]
                x[dept_name] := i[dept_name]
                x[name] := i[name]
                x[salary] := i[salary]
                x[building] := d[building]
                x[budget] := d[budget]
                include tuple x as part of result of instructor ⋈ department
             end
    end
end
```

**Figure 10.17** Procedure for computing join.

LRU is an acceptable replacement scheme in operating systems. However, a database system is able to predict the pattern of future references more accurately than an operating system. A user request to the database system involves several steps. The database system is often able to determine in advance which blocks will be needed by looking at each of the steps required to perform the user-requested operation. Thus, unlike operating systems, which must rely on the past to predict the future, database systems may have information regarding at least the short-term future.

To illustrate how information about future block access allows us to improve the LRU strategy, consider the processing of the SQL query:

```
select *
from instructor natural join department;
```

Assume that the strategy chosen to process this request is given by the pseudocode program shown in Figure 10.17. (We shall study other, more efficient, strategies in Chapter 12.)

Assume that the two relations of this example are stored in separate files. In this example, we can see that, once a tuple of *instructor* has been processed, that tuple is not needed again. Therefore, once processing of an entire block of *instructor* tuples is completed, that block is no longer needed in main memory, even though it has been used recently. The buffer manager should be instructed to free the space occupied by an *instructor* block as soon as the final tuple has been processed. This buffer-management strategy is called the **toss-immediate** strategy.

Now consider blocks containing *department* tuples. We need to examine every block of *department* tuples once for each tuple of the *instructor* relation. When

processing of a *department* block is completed, we know that that block will not be accessed again until all other *department* blocks have been processed. Thus, the most recently used *department* block will be the final block to be re-referenced, and the least recently used *department* block is the block that will be referenced next. This assumption set is the exact opposite of the one that forms the basis for the LRU strategy. Indeed, the optimal strategy for block replacement for the above procedure is the **most recently used** (**MRU**) strategy. If a *department* block must be removed from the buffer, the MRU strategy chooses the most recently used block (blocks are not eligible for replacement while they are being used).

For the MRU strategy to work correctly for our example, the system must pin the *department* block currently being processed. After the final *department* tuple has been processed, the block is unpinned, and it becomes the most recently used block.

In addition to using knowledge that the system may have about the request being processed, the buffer manager can use statistical information about the probability that a request will reference a particular relation. For example, the data dictionary that (as we will see in detail in Section 10.7) keeps track of the logical schema of the relations as well as their physical storage information is one of the most frequently accessed parts of the database. Thus, the buffer manager should try not to remove data-dictionary blocks from main memory, unless other factors dictate that it do so. In Chapter 11, we discuss indices for files. Since an index for a file may be accessed more frequently than the file itself, the buffer manager should, in general, not remove index blocks from main memory if alternatives are available.

The ideal database block-replacement strategy needs knowledge of the database operations—both those being performed and those that will be performed in the future. No single strategy is known that handles all the possible scenarios well. Indeed, a surprisingly large number of database systems use LRU, despite that strategy's faults. The practice questions and exercises explore alternative strategies.

The strategy that the buffer manager uses for block replacement is influenced by factors other than the time at which the block will be referenced again. If the system is processing requests by several users concurrently, the concurrency-control subsystem (Chapter 15) may need to delay certain requests, to ensure preservation of database consistency. If the buffer manager is given information from the concurrency-control subsystem indicating which requests are being delayed, it can use this information to alter its block-replacement strategy. Specifically, blocks needed by active (nondelayed) requests can be retained in the buffer at the expense of blocks needed by the delayed requests.

The crash-recovery subsystem (Chapter 16) imposes stringent constraints on block replacement. If a block has been modified, the buffer manager is not allowed to write back the new version of the block in the buffer to disk, since that would destroy the old version. Instead, the block manager must seek permission from the crash-recovery subsystem before writing out a block. The crash-recovery subsystem may demand that certain other blocks be force-output before it grants permission to the buffer manager to output the block requested. In Chapter 16,

we define precisely the interaction between the buffer manager and the crash-recovery subsystem.

## 10.9 Summary

- Several types of data storage exist in most computer systems. They are classified by the speed with which they can access data, by their cost per unit of data to buy the memory, and by their reliability. Among the media available are cache, main memory, flash memory, magnetic disks, optical disks, and magnetic tapes.
- Two factors determine the reliability of storage media: whether a power failure or system crash causes data to be lost, and what the likelihood is of physical failure of the storage device.
- We can reduce the likelihood of physical failure by retaining multiple copies of data. For disks, we can use mirroring. Or we can use more sophisticated methods based on redundant arrays of independent disks (RAID). By striping data across disks, these methods offer high throughput rates on large accesses; by introducing redundancy across disks, they improve reliability greatly. Several different RAID organizations are possible, each with different cost, performance, and reliability characteristics. RAID level 1 (mirroring) and RAID level 5 are the most commonly used.
- We can organize a file logically as a sequence of records mapped onto disk blocks. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure files so that they can accommodate multiple lengths for records. The slotted-page method is widely used to handle varying length records within a disk block.
- Since data are transferred between disk storage and main memory in units of a block, it is worthwhile to assign file records to blocks in such a way that a single block contains related records. If we can access several of the records we want with only one block access, we save disk accesses. Since disk accesses are usually the bottleneck in the performance of a database system, careful assignment of records to blocks can pay significant performance dividends.
- The data dictionary, also referred to as the system catalog, keeps track of metadata, that is data about data, such as relation names, attribute names and types, storage information, integrity constraints, and user information.
- One way to reduce the number of disk accesses is to keep as many blocks as possible in main memory. Since it is not possible to keep all blocks in main memory, we need to manage the allocation of the space available in main memory for the storage of blocks. The *buffer* is that part of main memory available for storage of copies of disk blocks. The subsystem responsible for the allocation of buffer space is called the *buffer manager*.

## Review Terms

- Physical storage media
  - Cache
  - Main memory
  - Flash memory
  - Magnetic disk
  - Optical storage
- Magnetic disk
  - Platter
  - Hard disks
  - Floppy disks
  - Tracks
  - Sectors
  - Read–write head
  - Disk arm
  - Cylinder
  - Disk controller
  - Checksums
  - Remapping of bad sectors
- Performance measures of disks
  - Access time
  - Seek time
  - Rotational latency
  - Data-transfer rate
  - Mean time to failure (MTTF)
- Disk block
- Optimization of disk-block access
  - Disk-arm scheduling
  - Elevator algorithm
  - File organization
  - Defragmenting
  - Nonvolatile write buffers
  - Nonvolatile random-access memory (NVRAM)
  - Log disk
- Redundant arrays of independent disks (RAID)
  - Mirroring
  - Data striping
  - Bit-level striping
  - Block-level striping
- RAID levels
  - Level 0 (block striping, no redundancy)
  - Level 1 (block striping, mirroring)
  - Level 3 (bit striping, parity)
  - Level 5 (block striping, distributed parity)
  - Level 6 (block striping, P + Q redundancy)
- Rebuild performance
- Software RAID
- Hardware RAID
- Hot swapping
- Tertiary storage
  - Optical disks
  - Magnetic tapes
  - Jukeboxes
- File
- File organization
  - File header

- Free list
- Variable-length records
  - Slotted-page structure
- Large objects
- Heap file organization
- Sequential file organization
- Hashing file organization
- Multitable clustering file organization
- Search key
- Data dictionary
- System catalog
- Buffer
  - Buffer manager
  - Pinned blocks
  - Forced output of blocks
- Buffer-replacement policies
  - Least recently used (LRU)
  - Toss-immediate
  - Most recently used (MRU)

## Practice Exercises

**10.1** Consider the data and parity-block arrangement on four disks depicted in Figure 10.18. The $B_i$s represent data blocks; the $P_i$s represent parity blocks. Parity block $P_i$ is the parity block for data blocks $B_{4i-3}$ to $B_{4i}$. What, if any, problem might this arrangement present?

**10.2** Flash storage:

a. How is the flash translation table, which is used to map logical page numbers to physical page numbers, created in memory?

b. Suppose you have a 64 gigabyte flash storage system, with a 4096 byte page size. How big would the flash translation table be, assuming each page has a 32 bit address, and the table is stored as an array.

c. Suppose how to reduce the size of the translation table if very often long ranges of consecutive logical page numbers are mapped to consecutive physical page numbers.

| Disk 1 | Disk 2 | Disk 3 | Disk 4 |
|---|---|---|---|
| $B_1$ | $B_2$ | $B_3$ | $B_4$ |
| $P_1$ | $B_5$ | $B_6$ | $B_7$ |
| $B_8$ | $P_2$ | $B_9$ | $B_{10}$ |
| ⋮ | ⋮ | ⋮ | ⋮ |

**Figure 10.18** Data and parity block arrangement.

**10.3** A power failure that occurs while a disk block is being written could result in the block being only partially written. Assume that partially written blocks can be detected. An atomic block write is one where either the disk block is fully written or nothing is written (i.e., there are no partial writes). Suggest schemes for getting the effect of atomic block writes with the following RAID schemes. Your schemes should involve work on recovery from failure.

a. RAID level 1 (mirroring)

b. RAID level 5 (block interleaved, distributed parity)

**10.4** Consider the deletion of record 5 from the file of Figure 10.6. Compare the relative merits of the following techniques for implementing the deletion:

a. Move record 6 to the space occupied by record 5, and move record 7 to the space occupied by record 6.

b. Move record 7 to the space occupied by record 5.

c. Mark record 5 as deleted, and move no records.

**10.5** Show the structure of the file of Figure 10.7 after each of the following steps:

a. Insert (24556, Turnamian, Finance, 98000).

b. Delete record 2.

c. Insert (34556, Thompson, Music, 67000).

**10.6** Consider the relations *section* and *takes*. Give an example instance of these two relations, with three sections, each of which has five students. Give a file structure of these relations that uses multitable clustering.

**10.7** Consider the following bitmap technique for tracking free space in a file. For each block in the file, two bits are maintained in the bitmap. If the block is between 0 and 30 percent full the bits are 00, between 30 and 60 percent the bits are 01, between 60 and 90 percent the bits are 10, and above 90 percent the bits are 11. Such bitmaps can be kept in memory even for quite large files.

a. Describe how to keep the bitmap up to date on record insertions and deletions.

b. Outline the benefit of the bitmap technique over free lists in searching for free space and in updating free space information.

**10.8** It is important to be able to quickly find out if a block is present in the buffer, and if so where in the buffer it resides. Given that database buffer sizes are very large, what (in-memory) data structure would you use for the above task?

**10.9** Give an example of a relational-algebra expression and a query-processing strategy in each of the following situations:

a. MRU is preferable to LRU.

b. LRU is preferable to MRU.

## Exercises

**10.10** List the physical storage media available on the computers you use routinely. Give the speed with which data can be accessed on each medium.

**10.11** How does the remapping of bad sectors by disk controllers affect data-retrieval rates?

**10.12** RAID systems typically allow you to replace failed disks without stopping access to the system. Thus, the data in the failed disk must be rebuilt and written to the replacement disk while the system is in operation. Which of the RAID levels yields the least amount of interference between the rebuild and ongoing disk accesses? Explain your answer.

**10.13** What is scrubbing, in the context of RAID systems, and why is scrubbing important?

**10.14** In the variable-length record representation, a null bitmap is used to indicate if an attribute has the null value.

a. For variable length fields, if the value is null, what would be stored in the offset and length fields?

b. In some applications, tuples have a very large number of attributes, most of which are null. Can you modify the record representation such that the only overhead for a null attribute is the single bit in the null bitmap.

**10.15** Explain why the allocation of records to blocks affects database-system performance significantly.

**10.16** If possible, determine the buffer-management strategy used by the operating system running on your local computer system and what mechanisms it provides to control replacement of pages. Discuss how the control on replacement that it provides would be useful for the implementation of database systems.

**10.17** List two advantages and two disadvantages of each of the following strategies for storing a relational database:

a. Store each relation in one file.

b. Store multiple relations (perhaps even the entire database) in one file.

**10.18** In the sequential file organization, why is an overflow *block* used even if there is, at the moment, only one overflow record?

**10.19** Give a normalized version of the *Index_metadata* relation, and explain why using the normalized version would result in worse performance.

**10.20** If you have data that should not be lost on disk failure, and the data are write intensive, how would you store the data?

**10.21** In earlier generation disks the number of sectors per track was the same across all tracks. Current generation disks have more sectors per track on outer tracks, and fewer sectors per track on inner tracks (since they are shorter in length). What is the effect of such a change on each of the three main indicators of disk speed?

**10.22** Standard buffer managers assume each block is of the same size and costs the same to read. Consider a buffer manager that, instead of LRU, uses the rate of reference to objects, that is, how often an object has been accessed in the last $n$ seconds. Suppose we want to store in the buffer objects of varying sizes, and varying read costs (such as Web pages, whose read cost depends on the site from which they are fetched). Suggest how a buffer manager may choose which block to evict from the buffer.

## Bibliographical Notes

Hennessy et al. [2006] is a popular textbook on computer architecture, which includes coverage of hardware aspects of translation look-aside buffers, caches, and memory-management units. Rosch [2003] presents an excellent overview of computer hardware, including extensive coverage of all types of storage technology such as magnetic disks, optical disks, tapes, and storage interfaces. Patterson [2004] provides a good discussion on how latency improvements have lagged behind bandwidth (transfer rate) improvements.

With the rapid increase in CPU speeds, cache memory located along with the CPU has become much faster than main memory. Although database systems do not control what data is kept in cache, there is an increasing motivation to organize data in memory and write programs in such a way that cache utilization is maximized. Work in this area includes Rao and Ross [2000], Ailamaki et al. [2001], Zhou and Ross [2004], Garcia and Korth [2005], and Cieslewicz et al. [2009].

The specifications of current-generation disk drives can be obtained from the Web sites of their manufacturers, such as Hitachi, LaCie, Iomega, Seagate, Maxtor, and Western Digital.

Discussions of redundant arrays of inexpensive disks (RAID) are presented by Patterson et al. [1988]. Chen et al. [1994] presents an excellent survey of RAID principles and implementation. Reed–Solomon codes are covered in Pless [1998].

Buffering data in mobile systems is discussed in Imielinski and Badrinath [1994], Imielinski and Korth [1996], and Chandrasekaran et al. [2003].

The storage structure of specific database systems, such as IBM DB2, Oracle, Microsoft SQL Server, and PostgreSQL are documented in their respective system manuals.

Buffer management is discussed in most operating-system texts, including in Silberschatz et al. [2008]. Chou and Dewitt [1985] presents algorithms for buffer management in database systems, and describes a performance evaluation.

CHAPTER 11

# Indexing and Hashing

Many queries reference only a small proportion of the records in a file. For example, a query like "Find all instructors in the Physics department" or "Find the total number of credits earned by the student with *ID* 22201" references only a fraction of the student records. It is inefficient for the system to read every tuple in the *instructor* relation to check if the *dept_name* value is "Physics". Likewise, it is inefficient to read the entire *student* relation just to find the one tuple for the *ID* "32556,". Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures that we associate with files.

## 11.1 Basic Concepts

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking. The words in the index are in sorted order, making it easy to find the word we want. Moreover, the index is much smaller than the book, further reducing the effort needed.

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

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

There are two basic kinds of indices:

- **Ordered indices**. Based on a sorted ordering of the values.

- **Hash indices**. Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a *hash function*.

We shall consider several techniques for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

- **Access types**: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time**: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time**: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead**: The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

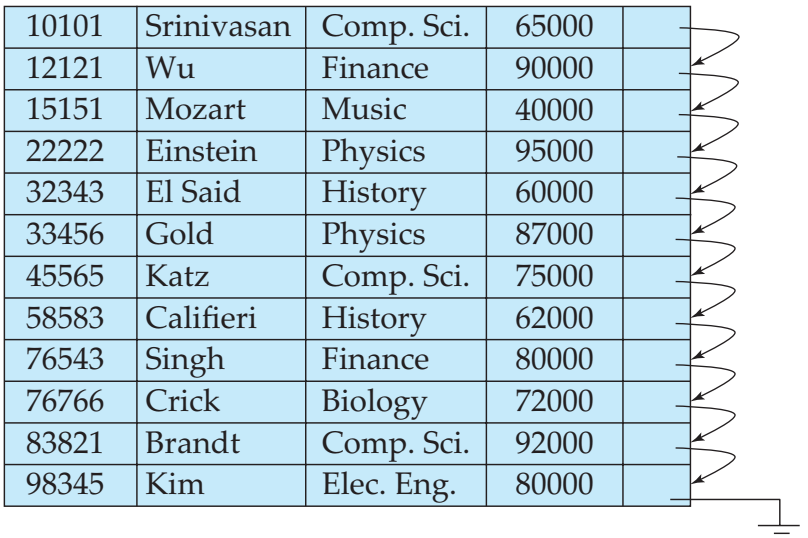
We often want to have more than one index for a file. For example, we may wish to search for a book by author, by subject, or by title.

An attribute or set of attributes used to look up records in a file is called a **search key**. Note that this definition of *key* differs from that used in *primary key*, *candidate key*, and *superkey*. This duplicate meaning for *key* is (unfortunately) well established in practice. Using our notion of a search key, we see that if there are several indices on a file, there are several search keys.

## 11.2 Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.

The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute such as the Dewey decimal number. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file. Clustering indices



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

**Figure 11.1** Sequential file for *instructor* records.

are also called **primary indices**; the term primary index may appear to denote an index on a primary key, but such indices can in fact be built on any search key. The search key of a clustering index is often the primary key, although that is not necessarily so. Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices**, or **secondary** indices. The terms "clustered" and "nonclustered" are often used in place of "clustering" and "nonclustering."

In Sections 11.2.1 through 11.2.3, we assume that all files are ordered sequentially on some search key. Such files, with a clustering index on the search key, are called **index-sequential files**. They represent one of the oldest index schemes used in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records. In Section 11.2.4 we cover secondary indices.

Figure 11.1 shows a sequential file of *instructor* records taken from our university example. In the example of Figure 11.1, the records are stored in sorted order of instructor *ID*, which is used as the search key.

### 11.2.1 Dense and Sparse Indices

An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

There are two types of ordered indices that we can use:

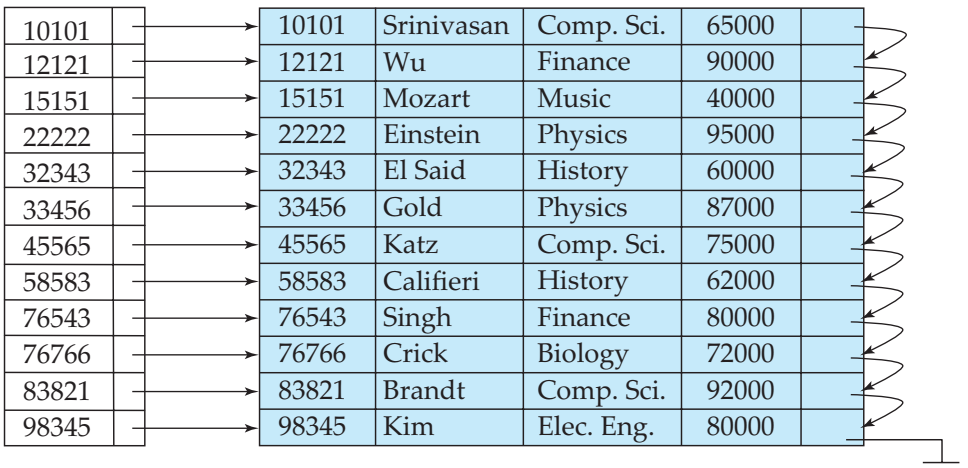



**Figure 11.2** Dense index.

- **Dense index**: In a dense index, an index entry appears for every search-key value in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key.

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

- **Sparse index**: In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index. As is true in dense indices, each index entry contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

Figures 11.2 and 11.3 show dense and sparse indices, respectively, for the *instructor* file. Suppose that we are looking up the record of instructor with *ID* "22222". Using the dense index of Figure 11.2, we follow the pointer directly to the desired record. Since *ID* is a primary key, there exists only one such record and the search is complete. If we are using the sparse index (Figure 11.3), we do not find an index entry for "22222". Since the last entry (in numerical order) before "22222" is "10101", we follow that pointer. We then read the *instructor* file in sequential order until we find the desired record.

Consider a (printed) dictionary. The header of each page lists the first word alphabetically on that page. The words at the top of each page of the book index together form a sparse index on the contents of the dictionary pages.

As another example, suppose that the search-key value is not not a primary key. Figure 11.4 shows a dense clustering index for the *instructor* file with the search key being *dept_name*. Observe that in this case the *instructor* file is sorted on the search key *dept_name*, instead of *ID*, otherwise the index on *dept_name* would be a nonclustering index. Suppose that we are looking up records for the History department. Using the dense index of Figure 11.4, we follow the pointer directly to the first History record. We process this record, and follow the pointer in that record to locate the next record in search-key (*dept_name*) order. We continue processing records until we encounter a record for a department other than History.

As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

There is a trade-off that the system designer must make between access time and space overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per block. The reason this design is a good trade-off is that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible. Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block (see Section 10.6.1), we minimize block accesses while keeping the size of the index (and thus our space overhead) as small as possible.

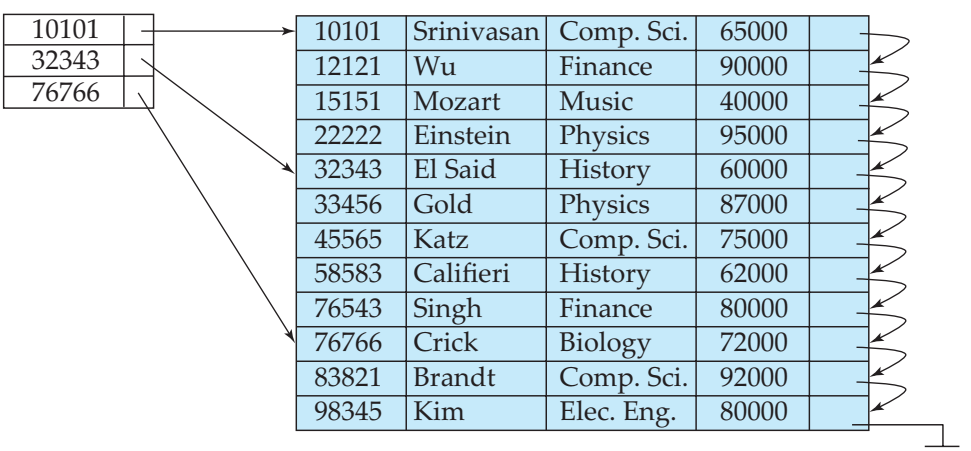



**Figure 11.3** Sparse index.

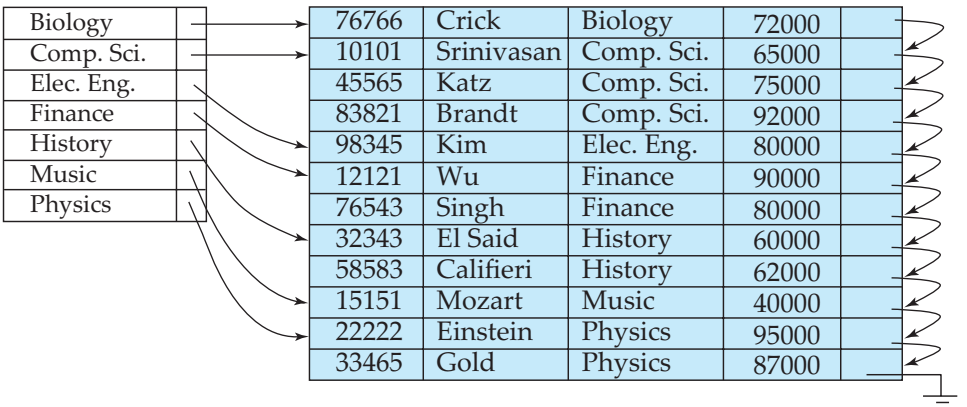



**Figure 11.4** Dense index with search key *dept_name*.

For the preceding technique to be fully general, we must consider the case where records for one search-key value occupy several blocks. It is easy to modify our scheme to handle this situation.

### 11.2.2 Multilevel Indices

Suppose we build a dense index on a relation with 1,000,000 tuples. Index entries are smaller than data records, so let us assume that 100 index entries fit on a 4 kilobyte block. Thus, our index occupies 10,000 blocks. If the relation instead had 100,000,000 tuples, the index would instead occupy 1,000,000 blocks, or 4 gigabytes of space. Such large indices are stored as sequential files on disk.

If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. (Even if an index is smaller than the main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory.) The search for an entry in the index then requires several disk-block reads.

Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index would occupy $b$ blocks, binary search requires as many as $\lceil \log_2(b) \rceil$ blocks to be read. ($\lceil x \rceil$ denotes the least integer that is greater than or equal to $x$; that is, we round upward.) For a 10,000-block index, binary search requires 14 block reads. On a disk system where a block read takes on average 10 milliseconds, the index search will take 140 milliseconds. This may not seem much, but we would be able to carry out only seven index searches a second, whereas a more efficient search mechanism would let us carry out far more searches per second, as we shall see shortly. Note that, if overflow blocks have been used, binary search is not possible. In that case, a sequential search is typically used, and that requires $b$ block reads, which will take even longer. Thus, the process of searching a large index may be costly.

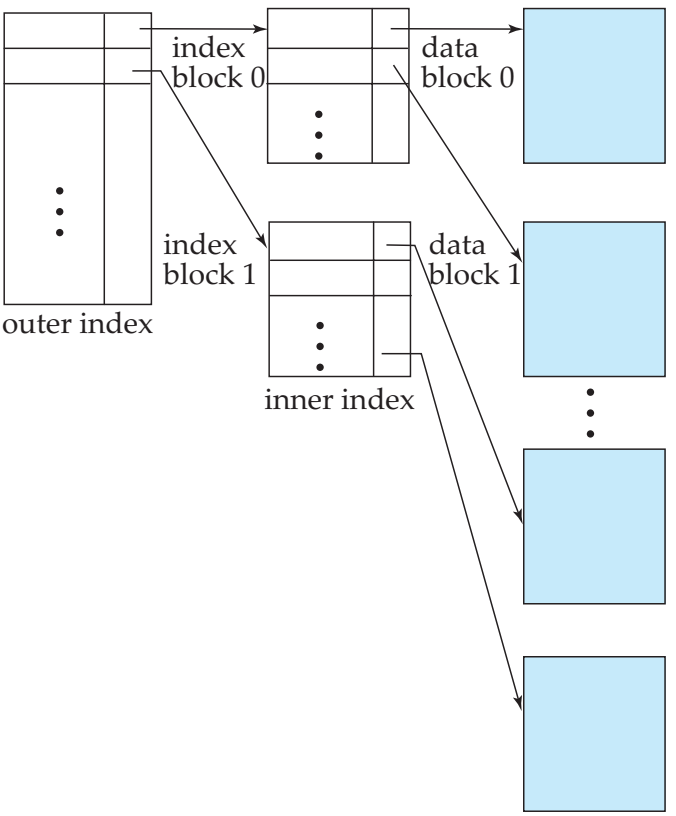



**Figure 11.5** Two-level sparse index.

To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse outer index on the original index, which we now call the inner index, as shown in Figure 11.5. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.

In our example, an inner index with 10,000 blocks would require 10,000 entries in the outer index, which would occupy just 100 blocks. If we assume that the outer index is already in main memory, we would read only one index block for a search using a multilevel index, rather than the 14 blocks we read with binary search. As a result, we can perform 14 times as many index searches per second.

If our file is extremely large, even the outer index may grow too large to fit in main memory. With a 100,000,000 tuple relation, the inner index would occupy

1,000,000 blocks, and the outer index occupies 10,000 blocks, or 40 megabytes. Since there are many demands on main memory, it may not be possible to reserve that much main memory just for this particular outer index. In such a case, we can create yet another level of index. Indeed, we can repeat this process as many times as necessary. Indices with two or more levels are called **multilevel** indices. Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search.[1]

Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing. We shall examine the relationship later, in Section 11.3.

### 11.2.3 Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file. Further, in case a record in the file is updated, any index whose search-key attribute is affected by the update must also be updated; for example, if the department of an instructor is changed, an index on the *dept_name* attribute of *instructor* must be updated correspondingly. Such a record update can be modeled as a deletion of the old record, followed by an insertion of the new value of the record, which results in an index deletion followed by an index insertion. As a result we only need to consider insertion and deletion on an index, and do not need to consider updates explicitly.

We first describe algorithms for updating single-level indices.

- **Insertion**. First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:
  - Dense indices:
    1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
    2. Otherwise the following actions are taken:
       a. If the index entry stores pointers to all records with the same search-key value, the system adds a pointer to the new record in the index entry.
       b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.
  - Sparse indices: We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in

[1] In the early days of disk-based indices, each level of the index corresponded to a unit of physical storage. Thus, we may have indices at the track, cylinder, and disk levels. Such a hierarchy does not make sense today since disk subsystems hide the physical details of disk storage, and the number of disks and platters per disk is very small compared to the number of cylinders or bytes per track.

search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

- **Deletion**. To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:
  - Dense indices:
    1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.
    2. Otherwise the following actions are taken:
       a. If the index entry stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index entry.
       b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.
  - Sparse indices:
    1. If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.
    2. Otherwise the system takes the following actions:
       a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
       b. Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.

Insertion and deletion algorithms for multilevel indices are a simple extension of the scheme just described. On deletion or insertion, the system updates the lowest-level index as described. As far as the second level is concerned, the lowest-level index is merely a file containing records—thus, if there is any change in the lowest-level index, the system updates the second-level index as described. The same technique applies to further levels of the index, if there are any.

### 11.2.4 Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing

only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

A secondary index on a candidate key looks just like a dense clustering index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from clustering indices. If the search key of a clustering index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.

In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file. Figure 11.6 shows the structure of a secondary index that uses an extra level of indirection on the *instructor* file, on the search key *salary*.

A sequential scan in clustering index order is efficient because records in the file are stored physically in the same order as the index order. However, we cannot (except in rare special cases) store a file physically ordered by both the search key of the clustering index and the search key of a secondary index.

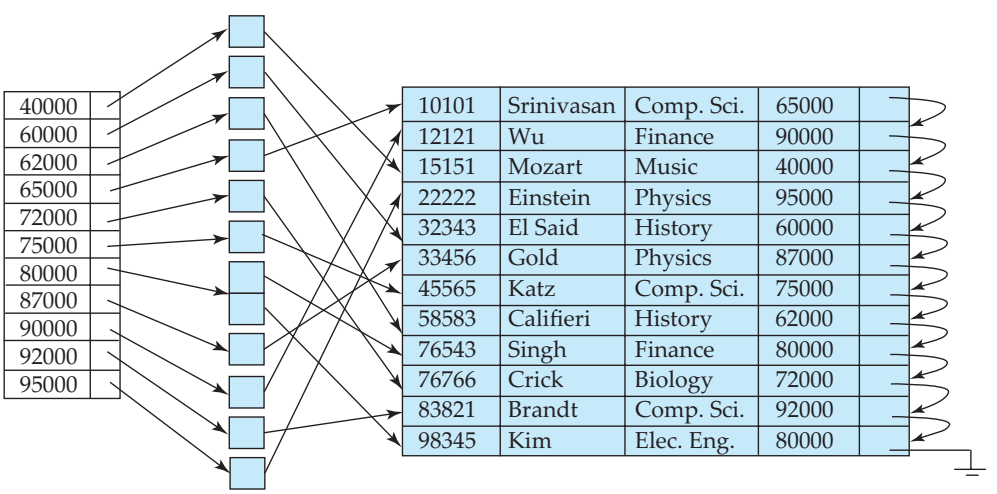



**Figure 11.6** Secondary index on *instructor* file, on noncandidate key *salary*.

**AUTOMATIC CREATION OF INDICES**

If a relation is declared to have a primary key, most database implementations automatically create an index on the primary key. Whenever a tuple is inserted into the relation, the index can be used to check that the primary key constraint is not violated (that is, there are no duplicates on the primary key value). Without the index on the primary key, whenever a tuple is inserted, the entire relation would have to be read to ensure that the primary-key constraint is satisfied.

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

The procedure described earlier for deletion and insertion can also be applied to secondary indices; the actions taken are those described for dense indices storing a pointer to every record in the file. If a file has multiple indices, whenever the file is modified, *every* index must be updated.

Secondary indices improve the performance of queries that use keys other than the search key of the clustering index. However, they impose a significant overhead on modification of the database. The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

### 11.2.5 Indices on Multiple Keys

Although the examples we have seen so far have had a single attribute in a search key, in general a search key can have more than one attribute. A search key containing more than one attribute is referred to as a **composite search key**. The structure of the index is the same as that of any other index, the only difference being that the search key is not a single attribute, but rather is a list of attributes. The search key can be represented as a tuple of values, of the form $(a_1, \ldots, a_n)$, where the indexed attributes are $A_1, \ldots, A_n$. The ordering of search-key values is the *lexicographic ordering*. For example, for the case of two attribute search keys, $(a_1, a_2) < (b_1, b_2)$ if either $a_1 < b_1$ or $a_1 = b_1$ and $a_2 < b_2$. Lexicographic ordering is basically the same as alphabetic ordering of words.

As an example, consider an index on the *takes* relation, on the composite search key (*course_id*, *semester*, *year*). Such an index would be useful to find all students who have registered for a particular course in a particular semester/year. An ordered index on a composite key can also be used to answer several other kinds of queries efficiently, as we shall see later in Section 11.5.2.

## 11.3 B+-Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans

through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.

The **B$^+$-tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B$^+$-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between $\lceil n/2 \rceil$ and $n$ children, where $n$ is fixed for a particular tree.

We shall see that the B$^+$-tree structure imposes performance overhead on insertion and deletion, and adds space overhead. The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, too, is acceptable given the performance benefits of the B$^+$-tree structure.

### 11.3.1 Structure of a B$^+$-Tree

A B$^+$-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure 11.7 shows a typical node of a B$^+$-tree. It contains up to $n - 1$ search-key values $K_1, K_2, \ldots, K_{n-1}$, and $n$ pointers $P_1, P_2, \ldots, P_n$. The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i < K_j$.

We consider first the structure of the **leaf nodes**. For $i = 1, 2, \ldots, n-1$, pointer $P_i$ points to a file record with search-key value $K_i$. Pointer $P_n$ has a special purpose that we shall discuss shortly.

Figure 11.8 shows one leaf node of a B$^+$-tree for the *instructor* file, in which we have chosen $n$ to be 4, and the search key is *name*.

Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to $n - 1$ values. We allow leaf nodes to contain as few as $\lceil (n - 1)/2 \rceil$ values. With $n = 4$ in our example B$^+$-tree, each leaf must contain at least 2 values, and at most 3 values.

The ranges of values in each leaf do not overlap, except if there are duplicate search-key values, in which case a value may be present in more than one leaf. Specifically, if $L_i$ and $L_j$ are leaf nodes and $i < j$, then every search-key value in $L_i$ is less than or equal to every search-key value in $L_j$. If the B$^+$-tree index is used as a dense index (as is usually the case) every search-key value must appear in some leaf node.

Now we can explain the use of the pointer $P_n$. Since there is a linear order on the leaves based on the search-key values that they contain, we use $P_n$ to chain


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

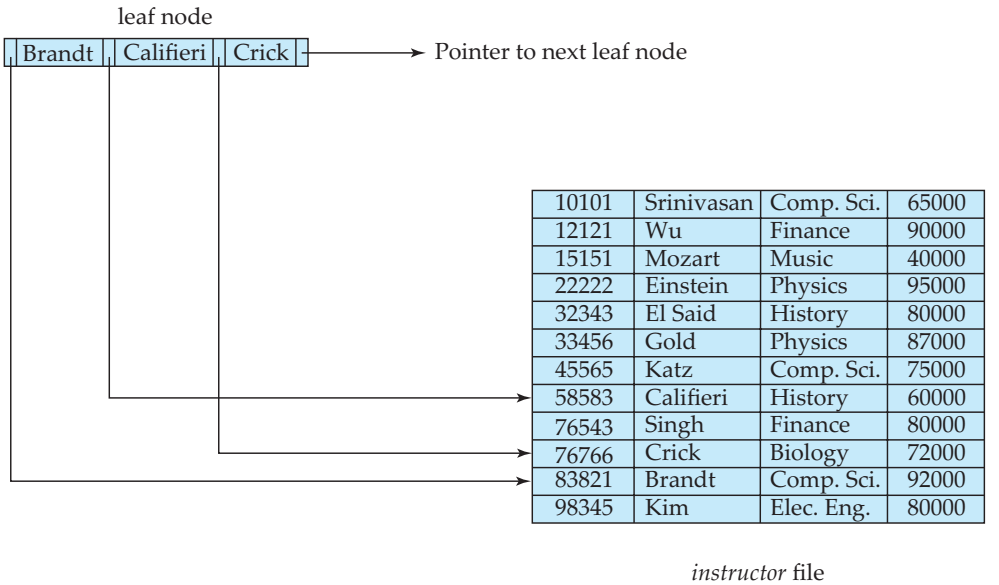



**Figure 11.8** A leaf node for *instructor* B+-tree index ($n = 4$).

together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.

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

Let us consider a node containing $m$ pointers ($m \leq n$). For $i = 2, 3, \ldots, m-1$, pointer $P_i$ points to the subtree that contains search-key values less than $K_i$ and greater than or equal to $K_{i-1}$. Pointer $P_m$ points to the part of the subtree that contains those key values greater than or equal to $K_{m-1}$, and pointer $P_1$ points to the part of the subtree that contains those search-key values less than $K_1$.

Unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B+-tree, for any $n$, that satisfies the preceding requirements.

Figure 11.9 shows a complete B+-tree for the *instructor* file (with $n = 4$). We have shown instructor names abbreviated to 3 characters in order to depict the tree clearly; in reality, the tree nodes would contain the full names. We have also omitted null pointers for simplicity; any pointer field in the figure that does not have an arrow is understood to have a null value.

Figure 11.10 shows another B+-tree for the *instructor* file, this time with $n = 6$. As before, we have abbreviated instructor names only for clarity of presentation.

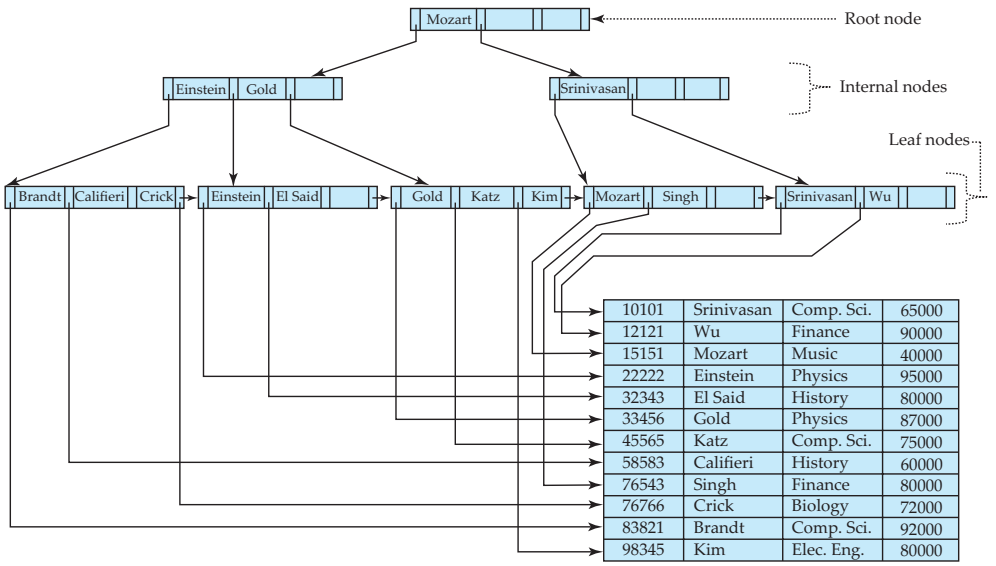



**Figure 11.9** B$^+$-tree for *instructor* file ($n = 4$).

Observe that the height of this tree is less than that of the previous tree, which had $n = 4$.

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

### 11.3.2 Queries on B$^+$-Trees

Let us consider how we process queries on a B$^+$-tree. Suppose that we wish to find records with a search-key value of $V$. Figure 11.11 presents pseudocode for a function `find()` to carry out this task.

Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest $i$ such that search-key value $K_i$ is greater

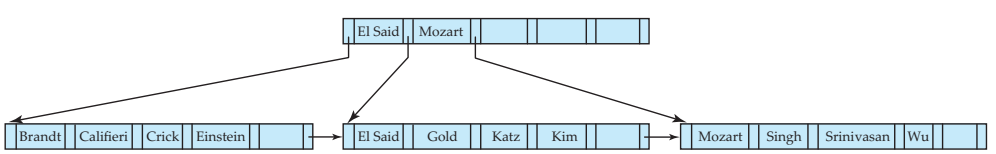



**Figure 11.10** B$^+$-tree for *instructor* file with $n = 6$.

```
function find(value V)
/* Returns leaf node C and index i such that C.P_i points to first record
* with search key value V */
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that V ≤ C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m
        end
        else if (V = C.K_i)
            then Set C = C.P_{i+1}
        else C = C.P_i /* V < C.K_i */
    end
    /* C is a leaf node */
    Let i be the least value such that K_i = V
    if there is such a value i
        then return (C, i)
        else return null ; /* No record with key value V exists*/


procedure printAll(value V)
/* prints all records with search key value V */
    Set done = false;
    Set (L, i) = find(V);
    if ((L, i) is null) return
    repeat
        repeat
            Print record pointed to by L.P_i
            Set i = i + 1
        until (i > number of keys in L or L.K_i > V)
        if (i > number of keys in L)
            then L = L.P_n
            else Set done = true;
    until (done or L is null)
```

**Figure 11.11** Querying a B+-tree.

than or equal to $V$. Suppose such a value is found; then, if $K_i$ is equal to $V$, the current node is set to the node pointed to by $P_{i+1}$, otherwise $K_i > V$, and the current node is set to the node pointed to by $P_i$. If no such value $K_i$ is found, then clearly $V > K_{m-1}$, where $P_m$ is the last nonnull pointer in the node. In this case the current node is set to that pointed to by $P_m$. The above procedure is repeated, traversing down the tree until a leaf node is reached.

At the leaf node, if there is a search-key value equal to $V$, let $K_i$ be the first such value; pointer $P_i$ directs us to a record with search-key value $K_i$. The function

then returns the leaf node $L$ and the index $i$. If no search-key with value $V$ is found in the leaf node, no record with key value $V$ exists in the relation, and function `find` returns null, to indicate failure.

If there is at most one record with a search key value $V$ (for example, if the index is on a primary key) the procedure that calls the `find` function simply uses the pointer $L.P_i$ to retrieve the record and is done. However, in case there may be more than one matching record, the remaining records also need to be fetched.

Procedure `printAll` shown in Figure 11.11 shows how to fetch all records with a specified search key $V$. The procedure first steps through the remaining keys in the node $L$, to find other records with search-key value $V$. If node $L$ contains at least one search-key value greater than $V$, then there are no more records matching $V$. Otherwise, the next leaf, pointed to by $P_n$ may contain further entries for $V$. The node pointed to by $P_n$ must then be searched to find further records with search-key value $V$. If the highest search-key value in the node pointed to by $P_n$ is also $V$, further leaves may have to be traversed, in order to find all matching records. The **repeat** loop in `printAll` carries out the task of traversing leaf nodes until all matching records have been found.

A real implementation would provide a version of `find` supporting an iterator interface similar to that provided by the JDBC `ResultSet`, which we saw in Section 5.1.1. Such an iterator interface would provide a method `next`(), which can be called repeatedly to fetch successive records with the specified search-key. The `next`() method would step through the entries at the leaf level, in a manner similar to `printAll`, but each call takes only one step, and records where it left off, so that successive calls `next` step through successive records. We omit details for simplicity, and leave the pseudocode for the iterator interface as an exercise for the interested reader.

$B^+$-trees can also be used to find all records with search key values in a specified range $(L, U)$. For example, with a $B^+$-tree on attribute *salary* of *instructor*, we can find all *instructor* records with salary in a specified range such as $(50000, 100000)$ (in other words, all salaries between 50000 and 100000). Such queries are called **range queries**. To execute such queries, we can create a procedure `printRange`$(L, U)$, whose body is the same as `printAll` except for these differences: `printRange` calls `find`$(L)$, instead of `find`$(V)$, and then steps through records as in procedure `printAll`, but with the stopping condition being that $L.K_i > U$, instead of $L.K_i > V$.

In processing a query, we traverse a path in the tree from the root to some leaf node. If there are $N$ records in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(N) \rceil$.

In practice, only a few nodes need to be accessed. Typically, a node is made to be the same size as a disk block, which is typically 4 kilobytes. With a search-key size of 12 bytes, and a disk-pointer size of 8 bytes, $n$ is around 200. Even with a more conservative estimate of 32 bytes for the search-key size, $n$ is around 100. With $n = 100$, if we have 1 million search-key values in the file, a lookup requires only $\lceil \log_{50}(1{,}000{,}000) \rceil = 4$ nodes to be accessed. Thus, at most four blocks need to be read from disk for the lookup. The root node of the tree is usually heavily accessed and is likely to be in the buffer, so typically only three or fewer blocks need to be read from disk.

An important difference between B+-tree structures and in-memory tree structures, such as binary trees, is the size of a node, and as a result, the height of the tree. In a binary tree, each node is small, and has at most two pointers. In a B+-tree, each node is large—typically a disk block—and a node can have a large number of pointers. Thus, B+-trees tend to be fat and short, unlike thin and tall binary trees. In a balanced binary tree, the path for a lookup can be of length $\lceil \log_2(N) \rceil$, where $N$ is the number of records in the file being indexed. With $N = 1{,}000{,}000$ as in the previous example, a balanced binary tree requires around 20 node accesses. If each node were on a different disk block, 20 block reads would be required to process a lookup, in contrast to the four block reads for the B+-tree. The difference is significant, since each block read could require a disk arm seek, and a block read together with the disk arm seek takes about 10 milliseconds on a typical disk.

### 11.3.3 Updates on B+-Trees

When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly. Recall that updates to a record can be modeled as a deletion of the old record followed by insertion of the updated record. Hence we only consider the case of insertion and deletion.

Insertion and deletion are more complicated than lookup, since it may be necessary to **split** a node that becomes too large as the result of an insertion, or to **coalesce** nodes (that is, combine nodes) if a node becomes too small (fewer than $\lceil n/2 \rceil$ pointers). Furthermore, when a node is split or a pair of nodes is combined, we must ensure that balance is preserved. To introduce the idea behind insertion and deletion in a B+-tree, we shall assume temporarily that nodes never become too large or too small. Under this assumption, insertion and deletion are performed as defined next.

- **Insertion**. Using the same technique as for lookup from the `find()` function (Figure 11.11), we first find the leaf node in which the search-key value would appear. We then insert an entry (that is, a search-key value and record pointer pair) in the leaf node, positioning it such that the search keys are still in order.
- **Deletion**. Using the same technique as for lookup, we find the leaf node containing the entry to be deleted, by performing a lookup on the search-key value of the deleted record; if there are multiple entries with the same search-key value, we search across all entries with the same search-key value until we find the entry that points to the record being deleted. We then remove the entry from the leaf node. All entries in the leaf node that are to the right of the deleted entry are shifted left by one position, so that there are no gaps in the entries after the entry is deleted.

We now consider the general case of insertion and deletion, dealing with node splitting and node coalescing.

.

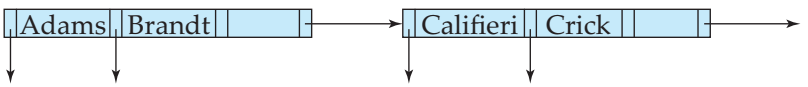



**Figure 11.12** Split of leaf node on insertion of "Adams"

#### 11.3.3.1 Insertion

We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the *instructor* relation, with the *name* value being Adams. We then need to insert an entry for "Adams" into the B$^+$-tree of Figure 11.9. Using the algorithm for lookup, we find that "Adams" should appear in the leaf node containing "Brandt", "Califieri", and "Crick." There is no room in this leaf to insert the search-key value "Adams." Therefore, the node is *split* into two nodes. Figure 11.12 shows the two leaf nodes that result from the split of the leaf node on inserting "Adams". The search-key values "Adams" and "Brandt" are in one leaf, and "Califieri" and "Crick" are in the other. In general, we take the $n$ search-key values (the $n - 1$ values in the leaf node plus the value being inserted), and put the first $\lceil n/2 \rceil$ in the existing node and the remaining values in a newly created node.

Having split a leaf node, we must insert the new leaf node into the B$^+$-tree structure. In our example, the new node has "Califieri" as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split. The B$^+$-tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

Splitting of a nonleaf node is a little different from splitting of a leaf node. Figure 11.14 shows the result of inserting a record with search key "Lamport" into the tree shown in Figure 11.13. The leaf node in which "Lamport" is to be inserted already has entries "Gold", "Katz", and "Kim", and as a result the leaf node has to be split. The new right-hand-side node resulting from the split contains the search-key values "Kim" and "Lamport". An entry (Kim, $n1$) must then be added

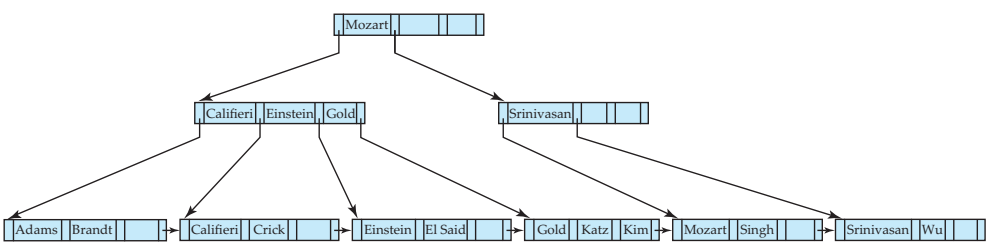



**Figure 11.13** Insertion of "Adams" into the B$^+$-tree of Figure 11.9.

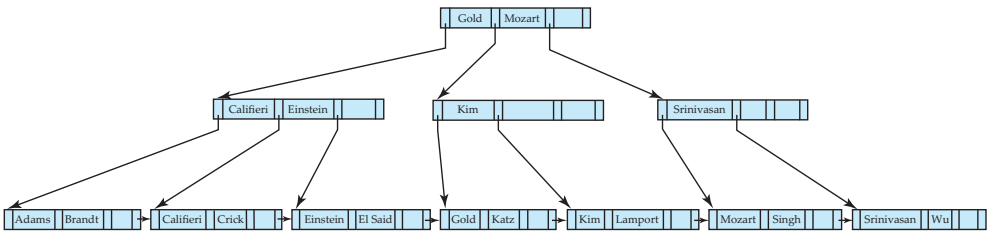



**Figure 11.14** Insertion of "Lamport" into the B+-tree of Figure 11.13.

to the parent node, where $n1$ is a pointer to the new node, However, there is no space in the parent node to add a new entry, and the parent node has to be split. To do so, the parent node is conceptually expanded temporarily, the entry added, and the overfull node is then immediately split.

When an overfull nonleaf node is split, the child pointers are divided among the original and the newly created nodes; in our example, the original node is left with the first three pointers, and the newly created node to the right gets the remaining two pointers. The search key values are, however, handled a little differently. The search key values that lie between the pointers moved to the right node (in our example, the value "Kim") are moved along with the pointers, while those that lie between the pointers that stay on the left (in our example, "Califieri" and "Einstein") remain undisturbed.

However, the search key value that lies between the pointers that stay on the left, and the pointers that move to the right node is treated differently. In our example, the search key value "Gold" lies between the three pointers that went to the left node, and the two pointers that went to the right node. The value "Gold" is not added to either of the split nodes. Instead, an entry (Gold, $n2$) is added to the parent node, where $n2$ is a pointer to the newly created node that resulted from the split. In this case, the parent node is the root, and it has enough space for the new entry.

The general technique for insertion into a B+-tree is to determine the leaf node $l$ into which insertion must occur. If a split results, insert the new node into the parent of node $l$. If this insertion causes a split, proceed recursively up the tree until either an insertion does not cause a split or a new root is created.

Figure 11.15 outlines the insertion algorithm in pseudocode. The procedure `insert` inserts a key-value pointer pair into the index, using two subsidiary procedures `insert_in_leaf` and `insert_in_parent`. In the pseudocode, $L$, $N$, $P$ and $T$ denote pointers to nodes, with $L$ being used to denote a leaf node. $L.K_i$ and $L.P_i$ denote the $i$th value and the $i$th pointer in node $L$, respectively; $T.K_i$ and $T.P_i$ are used similarly. The pseudocode also makes use of the function `parent`($N$) to find the parent of a node $N$. We can compute a list of nodes in the path from the root to the leaf while initially finding the leaf node, and can use it later to find the parent of any node in the path efficiently.

The procedure `insert_in_parent` takes as parameters $N$, $K'$, $N'$, where node $N$ was split into $N$ and $N'$, with $K'$ being the least value in $N'$. The procedure

```
procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than n − 1 key values)
        then insert_in_leaf (L, K, P)
        else begin /* L has n − 1 key values already, split it */
            Create node L′
            Copy L.P1 . . . L.Kn−1 to a block of memory T that can
                 hold n (pointer, key-value) pairs
            insert_in_leaf (T, K, P)
            Set L′.Pn = L.Pn; Set L.Pn = L′
            Erase L.P1 through L.Kn−1 from L
            Copy T.P1 through T.K⌈n/2⌉ from T into L starting at L.P1
            Copy T.P⌈n/2⌉+1 through T.Kn from T into L′ starting at L′.P1
            Let K′ be the smallest key-value in L′
            insert_in_parent(L, K′, L′)
        end

procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K1)
        then insert P, K into L just before L.P1
        else begin
            Let Ki be the highest value in L that is less than K
            Insert P, K into L just after T.Ki
        end

procedure insert_in_parent(node N, value K′, node N′)
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K′, N′   /* N and N′ are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K′, N′) in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K′, N′)
            Insert (K′, N′) into T just after N
            Erase all entries from P; Create node P′
            Copy T.P1 . . . T.P⌈n/2⌉ into P
            Let K″ = T.K⌈n/2⌉
            Copy T.P⌈n/2⌉+1 . . . T.Pn+1 into P′
            insert_in_parent(P, K″, P′)
        end
```

**Figure 11.15** Insertion of entry in a B$^+$-tree.

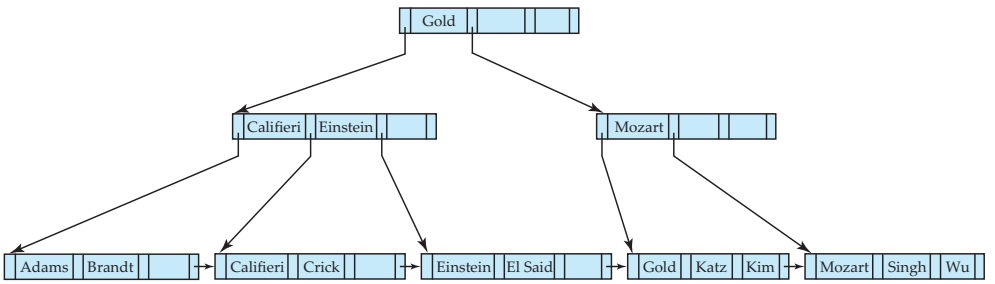



**Figure 11.16** Deletion of "Srinivasan" from the B+-tree of Figure 11.13.

modifies the parent of $N$ to record the split. The procedures `insert_into_index` and `insert_in_parent` use a temporary area of memory $T$ to store the contents of a node being split. The procedures can be modified to copy data from the node being split directly to the newly created node, reducing the time required for copying data. However, the use of the temporary space $T$ simplifies the procedures.

#### 11.3.3.2 Deletion

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete "Srinivasan" from the B+-tree of Figure 11.13. The resulting B+-tree appears in Figure 11.16. We now consider how the deletion is performed. We first locate the entry for "Srinivasan" by using our lookup algorithm. When we delete the entry for "Srinivasan" from its leaf node, the node is left with only one entry, "Wu". Since, in our example, $n = 4$ and $1 < \lceil (n-1)/2 \rceil$, we must either merge the node with a sibling node, or redistribute the entries between the nodes, to ensure that each node is at least half-full. In our example, the underfull node with the entry for "Wu" can be merged with its left sibling node. We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

In our example, the entry to be deleted is (Srinivasan, $n3$), where $n3$ is a pointer to the leaf containing "Srinivasan". (In this case the entry to be deleted in the nonleaf node happens to be the same value as that deleted from the leaf; that would not be the case for most deletions.) After deleting the above entry, the parent node, which had a search key value "Srinivasan" and two pointers, now has one pointer (the leftmost pointer in the node) and no search-key values. Since $1 < \lceil n/2 \rceil$ for $n = 4$, the parent node is underfull. (For larger $n$, a node that becomes underfull would still have some values as well as pointers.)

In this case, we look at a sibling node; in our example, the only sibling is the nonleaf node containing the search keys "Califieri", "Einstein", and "Gold". If possible, we try to coalesce the node with its sibling. In this case, coalescing is not possible, since the node and its sibling together have five pointers, against a maximum of four. The solution in this case is to **redistribute** the pointers between

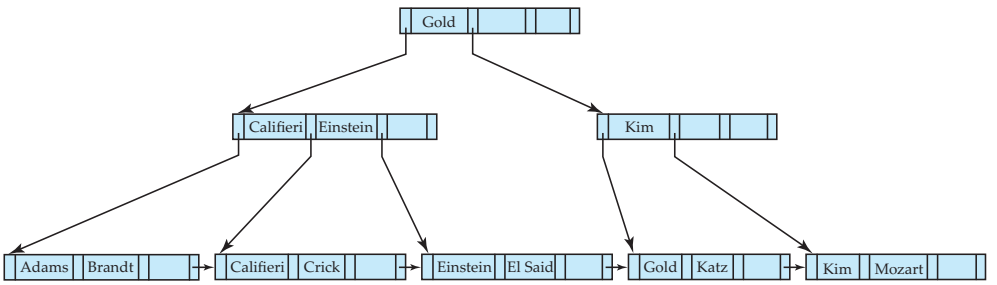



**Figure 11.17** Deletion of "Singh" and "Wu" from the B$^+$-tree of Figure 11.16.

the node and its sibling, such that each has at least $\lceil n/2 \rceil = 2$ child pointers. To do so, we move the rightmost pointer from the left sibling (the one pointing to the leaf node containing "Mozart") to the underfull right sibling. However, the underfull right sibling would now have two pointers, namely its leftmost pointer, and the newly moved pointer, with no value separating them. In fact, the value separating them is not present in either of the nodes, but is present in the parent node, between the pointers from the parent to the node and its sibling. In our example, the value "Mozart" separates the two pointers, and is present in the right sibling after the redistribution. Redistribution of the pointers also means that the value "Mozart" in the parent no longer correctly separates search-key values in the two siblings. In fact, the value that now correctly separates search-key values in the two sibling nodes is the value "Gold", which was in the left sibling before redistribution.

As a result, as can be seen in the B$^+$-tree in Figure 11.16, after redistribution of pointers between siblings, the value "Gold" has moved up into the parent, while the value that was there earlier, "Mozart", has moved down into the right sibling.

We next delete the search-key values "Singh" and "Wu" from the B$^+$-tree of Figure 11.16. The result is shown in Figure 11.17. The deletion of the first of these values does not make the leaf node underfull, but the deletion of the second value does. It is not possible to merge the underfull node with its sibling, so a redistribution of values is carried out, moving the search-key value "Kim" into the node containing "Mozart", resulting in the tree shown in Figure 11.17. The value separating the two siblings has been updated in the parent, from "Mozart" to "Kim".

Now we delete "Gold" from the above tree; the result is shown in Figure 11.18. This results in an underfull leaf, which can now be merged with its sibling. The resultant deletion of an entry from the parent node (the nonleaf node containing "Kim") makes the parent underfull (it is left with just one pointer). This time around, the parent node can be merged with its sibling. This merge results in the search-key value "Gold" moving down from the parent into the merged node. As a result of this merge, an entry is deleted from its parent, which happens to be the root of the tree. And as a result of that deletion, the root is left with only one child pointer and no search-key value, violating the condition that the root

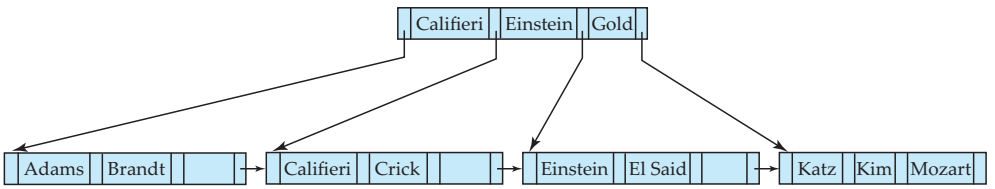



**Figure 11.18** Deletion of "Gold" from the B+-tree of Figure 11.17.

have at least two children. As a result, the root node is deleted and its sole child becomes the root, and the depth of the B+-tree has been decreased by 1.

It is worth noting that, as a result of deletion, a key value that is present in a nonleaf node of the B+-tree may not be present at any leaf of the tree. For example, in Figure 11.18, the value "Gold" has been deleted from the leaf level, but is still present in a nonleaf node.

In general, to delete a value in a B+-tree, we perform a lookup on the value and delete it. If the node is too small, we delete it from its parent. This deletion results in recursive application of the deletion algorithm until the root is reached, a parent remains adequately full after deletion, or redistribution is applied.

Figure 11.19 outlines the pseudocode for deletion from a B+-tree. The procedure `swap_variables`$(N, N')$ merely swaps the values of the (pointer) variables $N$ and $N'$; this swap has no effect on the tree itself. The pseudocode uses the condition "too few pointers/values." For nonleaf nodes, this criterion means less than $\lceil n/2 \rceil$ pointers; for leaf nodes, it means less than $\lceil (n-1)/2 \rceil$ values. The pseudocode redistributes entries by borrowing a single entry from an adjacent node. We can also redistribute entries by repartitioning entries equally between the two nodes. The pseudocode refers to deleting an entry $(K, P)$ from a node. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the pointer $P$ precedes the key value $K$. For nonleaf nodes, $P$ follows the key value $K$.

### 11.3.4 Nonunique Search Keys

If a relation can have more than one record containing the same search key value (that is, two or more records can have the same values for the indexed attributes), the search key is said to be a **nonunique search key**.

One problem with nonunique search keys is in the efficiency of record deletion. Suppose a particular search-key value occurs a large number of times, and one of the records with that search key is to be deleted. The deletion may have to search through a number of entries, potentially across multiple leaf nodes, to find the entry corresponding to the particular record being deleted.

A simple solution to this problem, used by most database systems, is to make search keys unique by creating a composite search key containing the original search key and another attribute, which together are unique across all records. The extra attribute can be a record-id, which is a pointer to the record, or any other attribute whose value is unique among all records with the same search-

```
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap_variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (K_i, P_i) pairs in N to N'; set N'.P_n = N.P_n
                delete_entry(parent(N), K', N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N' */
            if (N' is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that N'.P_m is the last pointer in N'
                    remove (N'.K_{m-1}, N'.P_m) from N'
                    insert (N'.P_m, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_{m-1}
                end
                else begin
                    let m be such that (N'.P_m, N'.K_m) is the last pointer/value
                        pair in N'
                    remove (N'.P_m, N'.K_m) from N'
                    insert (N'.P_m, N'.K_m) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_m
                end
            end
            else ... symmetric to the then case ...
        end
    end
```

**Figure 11.19** Deletion of entry from a B$^+$-tree.

key value. The extra attribute is called a **uniquifier** attribute. When a record is to be deleted, the composite search-key value is computed from the record, and then used to look up the index. Since the value is unique, the corresponding leaf-

level entry can be found with a single traversal from root to leaf, with no further accesses at the leaf level. As a result, record deletion can be done efficiently.

A search with the original search-key attribute simply ignores the value of the uniquifier attribute when comparing search-key values.

With nonunique search keys, our B+-tree structure stores each key value as many times as there are records containing that value. An alternative is to store each key value only once in the tree, and to keep a bucket (or list) of record pointers with a search-key value, to handle nonunique search keys. This approach is more space efficient since it stores the key value only once; however, it creates several complications when B+-trees are implemented. If the buckets are kept in the leaf node, extra code is needed to deal with variable-size buckets, and to deal with buckets that grow larger than the size of the leaf node. If the buckets are stored in separate blocks, an extra I/O operation may be required to fetch records. In addition to these problems, the bucket approach also has the problem of inefficiency for record deletion if a search-key value occurs a large number of times.

### 11.3.5 Complexity of B+-Tree Updates

Although insertion and deletion operations on B+-trees are complicated, they require relatively few I/O operations, which is an important benefit since I/O operations are expensive. It can be shown that the number of I/O operations needed in the worst case for an insertion is proportional to $\log_{\lceil n/2 \rceil}(N)$, where $n$ is the maximum number of pointers in a node, and $N$ is the number of records in the file being indexed.

The worst-case complexity of the deletion procedure is also proportional to $\log_{\lceil n/2 \rceil}(N)$, provided there are no duplicate values for the search key. If there are duplicate values, deletion may have to search across multiple records with the same search-key value to find the correct entry to be deleted, which can be inefficient. However, making the search key unique by adding a uniquifier attribute, as described in Section 11.3.4, ensures the worst-case complexity of deletion is the same even if the original search key is nonunique.

In other words, the cost of insertion and deletion operations in terms of I/O operations is proportional to the height of the B+-tree, and is therefore low. It is the speed of operation on B+-trees that makes them a frequently used index structure in database implementations.

In practice, operations on B+-trees result in fewer I/O operations than the worst-case bounds. With fanout of 100, and assuming accesses to leaf nodes are uniformly distributed, the parent of a leaf node is 100 times more likely to get accessed than the leaf node. Conversely, with the same fanout, the total number of nonleaf nodes in a B+-tree would be just a little more than 1/100th of the number of leaf nodes. As a result, with memory sizes of several gigabytes being common today, for B+-trees that are used frequently, even if the relation is very large it is quite likely that most of the nonleaf nodes are already in the database buffer when they are accessed. Thus, typically only one or two I/O operations are required to perform a lookup. For updates, the probability of a node split

occurring is correspondingly very small. Depending on the ordering of inserts, with a fanout of 100, only between 1 in 100 to 1 in 50 insertions will result in a node split, requiring more than one block to be written. As a result, on an average an insert will require just a little more than one I/O operation to write updated blocks.

Although $B^+$-trees only guarantee that nodes will be at least half full, if entries are inserted in random order, nodes can be expected to be more than two-thirds full on average. If entries are inserted in sorted order, on the other hand, nodes will be only half full. (We leave it as an exercise to the reader to figure out why nodes would be only half full in the latter case.)

## 11.4 $B^+$-Tree Extensions

In this section, we discuss several extensions and variations of the $B^+$-tree index structure.

### 11.4.1 $B^+$-Tree File Organization

As mentioned in Section 11.3, the main drawback of index-sequential file organization is the degradation of performance as the file grows: With growth, an increasing percentage of index entries and actual records become out of order, and are stored in overflow blocks. We solve the degradation of index lookups by using $B^+$-tree indices on the file. We solve the degradation problem for storing the actual records by using the leaf level of the $B^+$-tree to organize the blocks containing the actual records. We use the $B^+$-tree structure not only as an index, but also as an organizer for records in a file. In a **$B^+$-tree file organization**, the leaf nodes of the tree store records, instead of storing pointers to records. Figure 11.20 shows an example of a $B^+$-tree file organization. Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node. However, the leaf nodes are still required to be at least half full.

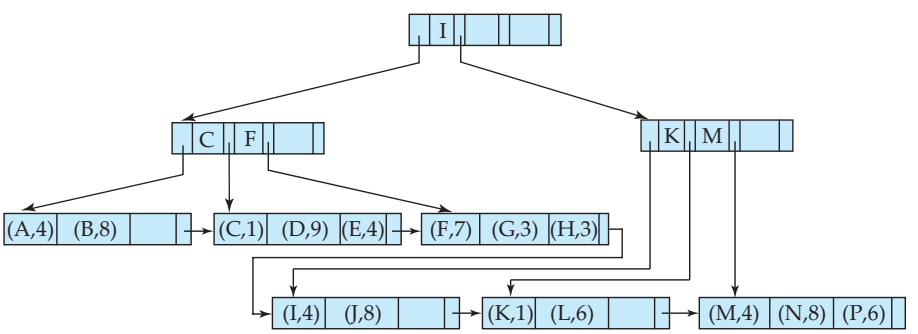



**Figure 11.20** $B^+$-tree file organization.

Insertion and deletion of records from a B+-tree file organization are handled in the same way as insertion and deletion of entries in a B+-tree index. When a record with a given key value $v$ is inserted, the system locates the block that should contain the record by searching the B+-tree for the largest key in the tree that is $\leq v$. If the block located has enough free space for the record, the system stores the record in the block. Otherwise, as in B+-tree insertion, the system splits the block in two, and redistributes the records in it (in the B+-tree–key order) to create space for the new record. The split propagates up the B+-tree in the normal fashion. When we delete a record, the system first removes it from the block containing it. If a block $B$ becomes less than half full as a result, the records in $B$ are redistributed with the records in an adjacent block $B'$. Assuming fixed-sized records, each block will hold at least one-half as many records as the maximum that it can hold. The system updates the nonleaf nodes of the B+-tree in the usual fashion.

When we use a B+-tree for file organization, space utilization is particularly important, since the space occupied by the records is likely to be much more than the space occupied by keys and pointers. We can improve the utilization of space in a B+-tree by involving more sibling nodes in redistribution during splits and merges. The technique is applicable to both leaf nodes and nonleaf nodes, and works as follows:

During insertion, if a node is full the system attempts to redistribute some of its entries to one of the adjacent nodes, to make space for a new entry. If this attempt fails because the adjacent nodes are themselves full, the system splits the node, and splits the entries evenly among one of the adjacent nodes and the two nodes that it obtained by splitting the original node. Since the three nodes together contain one more record than can fit in two nodes, each node will be about two-thirds full. More precisely, each node will have at least $\lfloor 2n/3 \rfloor$ entries, where $n$ is the maximum number of entries that the node can hold. ($\lfloor x \rfloor$ denotes the greatest integer that is less than or equal to $x$; that is, we drop the fractional part, if any.)

During deletion of a record, if the occupancy of a node falls below $\lfloor 2n/3 \rfloor$, the system attempts to borrow an entry from one of the sibling nodes. If both sibling nodes have $\lfloor 2n/3 \rfloor$ records, instead of borrowing an entry, the system redistributes the entries in the node and in the two siblings evenly between two of the nodes, and deletes the third node. We can use this approach because the total number of entries is $3\lfloor 2n/3 \rfloor - 1$, which is less than $2n$. With three adjacent nodes used for redistribution, each node can be guaranteed to have $\lfloor 3n/4 \rfloor$ entries. In general, if $m$ nodes ($m - 1$ siblings) are involved in redistribution, each node can be guaranteed to contain at least $\lfloor (m - 1)n/m \rfloor$ entries. However, the cost of update becomes higher as more sibling nodes are involved in the redistribution.

Note that in a B+-tree index or file organization, leaf nodes that are adjacent to each other in the tree may be located at different places on disk. When a file organization is newly created on a set of records, it is possible to allocate blocks that are mostly contiguous on disk to leaf nodes that are contiguous in the tree. Thus a sequential scan of leaf nodes would correspond to a mostly sequential scan on disk. As insertions and deletions occur on the tree, sequentiality is increasingly

lost, and sequential access has to wait for disk seeks increasingly often. An index rebuild may be required to restore sequentiality.

$B^+$-tree file organizations can also be used to store large objects, such as SQL clobs and blobs, which may be larger than a disk block, and as large as multiple gigabytes. Such large objects can be stored by splitting them into sequences of smaller records that are organized in a $B^+$-tree file organization. The records can be sequentially numbered, or numbered by the byte offset of the record within the large object, and the record number can be used as the search key.

### 11.4.2 Secondary Indices and Record Relocation

Some file organizations, such as the $B^+$-tree file organization, may change the location of records even when the records have not been updated. As an example, when a leaf node is split in a $B^+$-tree file organization, a number of records are moved to a new node. In such cases, all secondary indices that store pointers to the relocated records would have to be updated, even though the values in the records may not have changed. Each leaf node may contain a fairly large number of records, and each of them may be in different locations on each secondary index. Thus a leaf-node split may require tens or even hundreds of I/O operations to update all affected secondary indices, making it a very expensive operation.

A widely used solution for this problem is as follows: In secondary indices, in place of pointers to the indexed records, we store the values of the primary-index search-key attributes. For example, suppose we have a primary index on the attribute *ID* of relation *instructor*; then a secondary index on *dept_name* would store with each department name a list of instructor's *ID* values of the corresponding records, instead of storing pointers to the records.

Relocation of records because of leaf-node splits then does not require any update on any such secondary index. However, locating a record using the secondary index now requires two steps: First we use the secondary index to find the primary-index search-key values, and then we use the primary index to find the corresponding records.

The above approach thus greatly reduces the cost of index update due to file reorganization, although it increases the cost of accessing data using a secondary index.

### 11.4.3 Indexing Strings

Creating $B^+$-tree indices on string-valued attributes raises two problems. The first problem is that strings can be of variable length. The second problem is that strings can be long, leading to a low fanout and a correspondingly increased tree height.

With variable-length search keys, different nodes can have different fanouts even if they are full. A node must then be split if it is full, that is, there is no space to add a new entry, regardless of how many search entries it has. Similarly, nodes can be merged or entries redistributed depending on what fraction of the space in the nodes is used, instead of being based on the maximum number of entries that the node can hold.

The fanout of nodes can be increased by using a technique called **prefix compression**. With prefix compression, we do not store the entire search key value at nonleaf nodes. We only store a prefix of each search key value that is sufficient to distinguish between the key values in the subtrees that it separates. For example, if we had an index on names, the key value at a nonleaf node could be a prefix of a name; it may suffice to store "Silb" at a nonleaf node, instead of the full "Silberschatz" if the closest values in the two subtrees that it separates are, say, "Silas" and "Silver" respectively.

### 11.4.4 Bulk Loading of B+-Tree Indices

As we saw earlier, insertion of a record in a B+-tree requires a number of I/O operations that in the worst case is proportional to the height of the tree, which is usually fairly small (typically five or less, even for large relations).

Now consider the case where a B+-tree is being built on a large relation. Suppose the relation is significantly larger than main memory, and we are constructing a nonclustering index on the relation such that the index is also larger than main memory. In this case, as we scan the relation and add entries to the B+-tree, it is quite likely that each leaf node accessed is not in the database buffer when it is accessed, since there is no particular ordering of the entries. With such randomly ordered accesses to blocks, each time an entry is added to the leaf, a disk seek will be required to fetch the block containing the leaf node. The block will probably be evicted from the disk buffer before another entry is added to the block, leading to another disk seek to write the block back to disk. Thus a random read and a random write operation may be required for each entry inserted.

For example, if the relation has 100 million records, and each I/O operation takes about 10 milliseconds, it would take at least 1 million seconds to build the index, counting only the cost of reading leaf nodes, not even counting the cost of writing the updated nodes back to disk. This is clearly a very large amount of time; in contrast, if each record occupies 100 bytes, and the disk subsystem can transfer data at 50 megabytes per second, it would take just 200 seconds to read the entire relation.

Insertion of a large number of entries at a time into an index is referred to as **bulk loading** of the index. An efficient way to perform bulk loading of an index is as follows. First, create a temporary file containing index entries for the relation, then sort the file on the search key of the index being constructed, and finally scan the sorted file and insert the entries into the index. There are efficient algorithms for sorting large relations, which are described later in Section 12.4, which can sort even a large file with an I/O cost comparable to that of reading the file a few times, assuming a reasonable amount of main memory is available.

There is a significant benefit to sorting the entries before inserting them into the B+-tree. When the entries are inserted in sorted order, all entries that go to a particular leaf node will appear consecutively, and the leaf needs to be written out only once; nodes will never have to be read from disk during bulk load, if the B+-tree was empty to start with. Each leaf node will thus incur only one I/O operation even though many entries may be inserted into the node. If each leaf

contains 100 entries, the leaf level will contain 1 million nodes, resulting in only 1 million I/O operations for creating the leaf level. Even these I/O operations can be expected to be sequential, if successive leaf nodes are allocated on successive disk blocks, and few disk seeks would be required. With current disks, 1 millisecond per block is a reasonable estimate for mostly sequential I/O operations, in contrast to 10 milliseconds per block for random I/O operations.

We shall study the cost of sorting a large relation later, in Section 12.4, but as a rough estimate, the index which would have taken a million seconds to build otherwise, can be constructed in well under 1000 seconds by sorting the entries before inserting them into the $B^+$-tree, in contrast to more than 1,000,000 seconds for inserting in random order.

If the $B^+$-tree is initially empty, it can be constructed faster by building it bottom-up, from the leaf level, instead of using the usual insert procedure. In **bottom-up $B^+$-tree construction**, after sorting the entries as we just described, we break up the sorted entries into blocks, keeping as many entries in a block as can fit in the block; the resulting blocks form the leaf level of the $B^+$-tree. The minimum value in each block, along with the pointer to the block, is used to create entries in the next level of the $B^+$-tree, pointing to the leaf blocks. Each further level of the tree is similarly constructed using the minimum values associated with each node one level below, until the root is created. We leave details as an exercise for the reader.

Most database systems implement efficient techniques based on sorting of entries, and bottom-up construction, when creating an index on a relation, although they use the normal insertion procedure when tuples are added one at a time to a relation with an existing index. Some database systems recommend that if a very large number of tuples are added at once to an already existing relation, indices on the relation (other than any index on the primary key) should be dropped, and then re-created after the tuples are inserted, to take advantage of efficient bulk-loading techniques.

### 11.4.5 B-Tree Index Files

**B-tree indices** are similar to $B^+$-tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. In the $B^+$-tree of Figure 11.13, the search keys "Califieri", "Einstein", "Gold", "Mozart", and "Srinivasan" appear in nonleaf nodes, in addition to appearing in the leaf nodes. Every search-key value appears in some leaf node; several are repeated in nonleaf nodes.

A B-tree allows search-key values to appear only once (if they are unique), unlike a $B^+$-tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node. Figure 11.21 shows a B-tree that represents the same search keys as the $B^+$-tree of Figure 11.13. Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding $B^+$-tree index. However, since search keys that appear in nonleaf nodes appear nowhere else in the B-tree, we are forced to include an additional

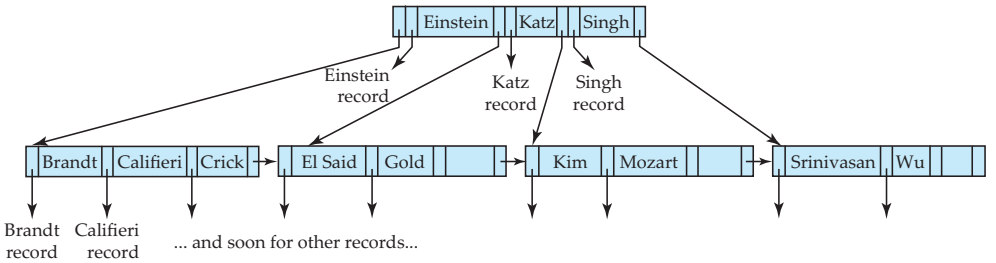



**Figure 11.21** B-tree equivalent of B+-tree in Figure 11.13.

pointer field for each search key in a nonleaf node. These additional pointers point to either file records or buckets for the associated search key.

It is worth noting that many database system manuals, articles in industry literature, and industry professionals use the term B-tree to refer to the data structure that we call the B+-tree. In fact, it would be fair to say that in current usage, the term B-tree is assumed to be synonymous with B+-tree. However, in this book we use the terms B-tree and B+-tree as they were originally defined, to avoid confusion between the two data structures.

A generalized B-tree leaf node appears in Figure 11.22a; a nonleaf node appears in Figure 11.22b. Leaf nodes are the same as in B+-trees. In nonleaf nodes, the pointers $P_i$ are the tree pointers that we used also for B+-trees, while the pointers $B_i$ are bucket or file-record pointers. In the generalized B-tree in the figure, there are $n-1$ keys in the leaf node, but there are $m-1$ keys in the nonleaf node. This discrepancy occurs because nonleaf nodes must include pointers $B_i$, thus reducing the number of search keys that can be held in these nodes. Clearly, $m < n$, but the exact relationship between $m$ and $n$ depends on the relative size of search keys and pointers.

The number of nodes accessed in a lookup in a B-tree depends on where the search key is located. A lookup on a B+-tree requires traversal of a path from the root of the tree to some leaf node. In contrast, it is sometimes possible to find the desired value in a B-tree before reaching a leaf node. However, roughly $n$ times as many keys are stored in the leaf level of a B-tree as in the nonleaf levels, and, since $n$ is typically large, the benefit of finding certain values early is relatively

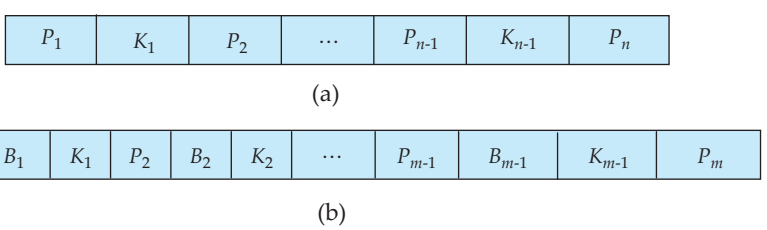



**Figure 11.22** Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

small. Moreover, the fact that fewer search keys appear in a nonleaf B-tree node, compared to $B^+$-trees, implies that a B-tree has a smaller fanout and therefore may have depth greater than that of the corresponding $B^+$-tree. Thus, lookup in a B-tree is faster for some search keys but slower for others, although, in general, lookup time is still proportional to the logarithm of the number of search keys.

Deletion in a B-tree is more complicated. In a $B^+$-tree, the deleted entry always appears in a leaf. In a B-tree, the deleted entry may appear in a nonleaf node. The proper value must be selected as a replacement from the subtree of the node containing the deleted entry. Specifically, if search key $K_i$ is deleted, the smallest search key appearing in the subtree of pointer $P_{i+1}$ must be moved to the field formerly occupied by $K_i$. Further actions need to be taken if the leaf node now has too few entries. In contrast, insertion in a B-tree is only slightly more complicated than is insertion in a $B^+$-tree.

The space advantages of B-trees are marginal for large indices, and usually do not outweigh the disadvantages that we have noted. Thus, pretty much all database-system implementations use the $B^+$-tree data structure, even if (as we discussed earlier) they refer to the data structure as a B-tree.

### 11.4.6 Flash Memory

In our description of indexing so far, we have assumed that data are resident on magnetic disks. Although this assumption continues to be true for the most part, flash memory capacities have grown significantly, and the cost of flash memory per gigabyte has dropped equally significantly, making flash memory storage a serious contender for replacing magnetic-disk storage for many applications. A natural question is, how would this change affect the index structure.

Flash-memory storage is structured as blocks, and the $B^+$-tree index structure can be used for flash-memory storage. The benefit of the much faster access speeds is clear for index lookups. Instead of requiring an average of 10 milliseconds to seek to and read a block, a random block can be read in about a microsecond from flash-memory. Thus lookups run significantly faster than with disk-based data. The optimum $B^+$-tree node size for flash-memory is typically smaller than that with disk.

The only real drawback with flash memory is that it does not permit in-place updates to data at the physical level, although it appears to do so logically. Every update turns into a copy+write of an entire flash-memory block, requiring the old copy of the block to be erased subsequently; a block erase takes about 1 millisecond. There is ongoing research aimed at developing index structures that can reduce the number of block erases. Meanwhile, standard $B^+$-tree indices can continue to be used even on flash-memory storage, with acceptable update performance, and significantly improved lookup performance compared to disk storage.

## 11.5 Multiple-Key Access

Until now, we have assumed implicitly that only one index on one attribute is used to process a query on a relation. However, for certain types of queries, it is

advantageous to use multiple indices if they exist, or to use an index built on a multiattribute search key.

### 11.5.1 Using Multiple Single-Key Indices

Assume that the *instructor* file has two indices: one for *dept_name* and one for *salary*. Consider the following query: "Find all instructors in the Finance department with salary equal to $80,000." We write

```
select ID
from instructor
where dept_name ='Finance' and salary= 80000;
```

There are three strategies possible for processing this query:

1. Use the index on *dept_name* to find all records pertaining to the Finance department. Examine each such record to see whether *salary*= 80000.
2. Use the index on *salary* to find all records pertaining to instructors with salary of $80,000. Examine each such record to see whether the department name is "Finance".
3. Use the index on *dept_name* to find *pointers* to all records pertaining to the Finance department. Also, use the index on *salary* to find pointers to all records pertaining to instructors with a salary of $80,000. Take the intersection of these two sets of pointers. Those pointers that are in the intersection point to records pertaining to instructors of the Finance department and with salary of $80,000.

The third strategy is the only one of the three that takes advantage of the existence of multiple indices. However, even this strategy may be a poor choice if all of the following hold:

- There are many records pertaining to the Finance department.
- There are many records pertaining to instructors with a salary of $80,000.
- There are only a few records pertaining to *both* the Finance department and instructors with a salary of $80,000.

If these conditions hold, we must scan a large number of pointers to produce a small result. An index structure called a "bitmap index" can in some cases greatly speed up the intersection operation used in the third strategy. Bitmap indices are outlined in Section 11.9.

### 11.5.2 Indices on Multiple Keys

An alternative strategy for this case is to create and use an index on a composite search key (*dept_name*, *salary*)—that is, the search key consisting of the department name concatenated with the instructor salary.

We can use an ordered ($B^+$-tree) index on the above composite search key to answer efficiently queries of the form

**select** *ID*
**from** *instructor*
**where** *dept_name* = 'Finance' **and** *salary*= 80000;

Queries such as the following query, which specifies an equality condition on the first attribute of the search key (*dept_name*) and a range on the second attribute of the search key (*salary*), can also be handled efficiently since they correspond to a range query on the search attribute.

**select** *ID*
**from** *instructor*
**where** *dept_name* = 'Finance' **and** *salary*< 80000;

We can even use an ordered index on the search key (*dept_name*, *salary*) to answer the following query on only one attribute efficiently:

**select** *ID*
**from** *instructor*
**where** *dept_name* = 'Finance';

An equality condition *dept_name* = "Finance" is equivalent to a range query on the range with lower end (Finance, $-\infty$) and upper end (Finance, $+\infty$). Range queries on just the *dept_name* attribute can be handled in a similar manner.

The use of an ordered-index structure on a composite search key, however, has a few shortcomings. As an illustration, consider the query

**select** *ID*
**from** *instructor*
**where** *dept_name* < 'Finance' **and** *salary*< 80000;

We can answer this query by using an ordered index on the search key (*dept_name*, *salary*): For each value of *dept_name* that is less than "Finance" in alphabetic order, the system locates records with a *salary* value of 80000. However, each record is likely to be in a different disk block, because of the ordering of records in the file, leading to many I/O operations.

The difference between this query and the previous two queries is that the condition on the first attribute (*dept_name*) is a comparison condition, rather than

an equality condition. The condition does not correspond to a range query on the search key.

To speed the processing of general composite search-key queries (which can involve one or more comparison operations), we can use several special structures. We shall consider *bitmap indices* in Section 11.9. There is another structure, called the *R-tree*, that can be used for this purpose. The R-tree is an extension of the $B^+$-tree to handle indexing on multiple dimensions. Since the R-tree is used primarily with geographical data types, we describe the structure in Chapter 25.

### 11.5.3 Covering Indices

**Covering indices** are indices that store the values of some attributes (other than the search-key attributes) along with the pointers to the record. Storing extra attribute values is useful with secondary indices, since they allow us to answer some queries using just the index, without even looking up the actual records.

For example, suppose that we have a nonclustering index on the *ID* attribute of the *instructor* relation. If we store the value of the *salary* attribute along with the record pointer, we can answer queries that require the salary (but not the other attribute, *dept_name*) without accessing the *instructor* record.

The same effect could be obtained by creating an index on the search key (*ID*, *salary*), but a covering index reduces the size of the search key, allowing a larger fanout in the nonleaf nodes, and potentially reducing the height of the index.

## 11.6 Static Hashing

One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations. File organizations based on the technique of **hashing** allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices. We study file organizations and indices based on hashing in the following sections.

In our description of hashing, we shall use the term **bucket** to denote a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.

Formally, let $K$ denote the set of all search-key values, and let $B$ denote the set of all bucket addresses. A **hash function** $h$ is a function from $K$ to $B$. Let $h$ denote a hash function.

To insert a record with search key $K_i$, we compute $h(K_i)$, which gives the address of the bucket for that record. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket.

To perform a lookup on a search-key value $K_i$, we simply compute $h(K_i)$, then search the bucket with that address. Suppose that two search keys, $K_5$ and $K_7$, have the same hash value; that is, $h(K_5) = h(K_7)$. If we perform a lookup on $K_5$, the bucket $h(K_5)$ contains records with search-key values $K_5$ and records

with search-key values $K_7$. Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want.

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

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

### 11.6.1 Hash Functions

The worst possible hash function maps all search-key values to the same bucket. Such a function is undesirable because all the records have to be kept in the same bucket. A lookup has to examine every such record to find the one desired. An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.

Since we do not know at design time precisely which search-key values will be stored in the file, we want to choose a hash function that assigns search-key values to buckets in such a way that the distribution has these qualities:

- The distribution is *uniform*. That is, the hash function assigns each bucket the same number of search-key values from the set of *all* possible search-key values.
- The distribution is *random*. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key values. More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.

As an illustration of these principles, let us choose a hash function for the *instructor* file using the search key *dept_name*. The hash function that we choose must have the desirable properties not only on the example *instructor* file that we have been using, but also on an *instructor* file of realistic size for a large university with many departments.

Assume that we decide to have 26 buckets, and we define a hash function that maps names beginning with the $i$th letter of the alphabet to the $i$th bucket. This hash function has the virtue of simplicity, but it fails to provide a uniform distribution, since we expect more names to begin with such letters as B and R than Q and X, for example.

Now suppose that we want a hash function on the search key *salary*. Suppose that the minimum salary is $30,000 and the maximum salary is $130,000, and we use a hash function that divides the values into 10 ranges, $30,000–$40,000, $40,001–$50,000 and so on. The distribution of search-key values is uniform (since

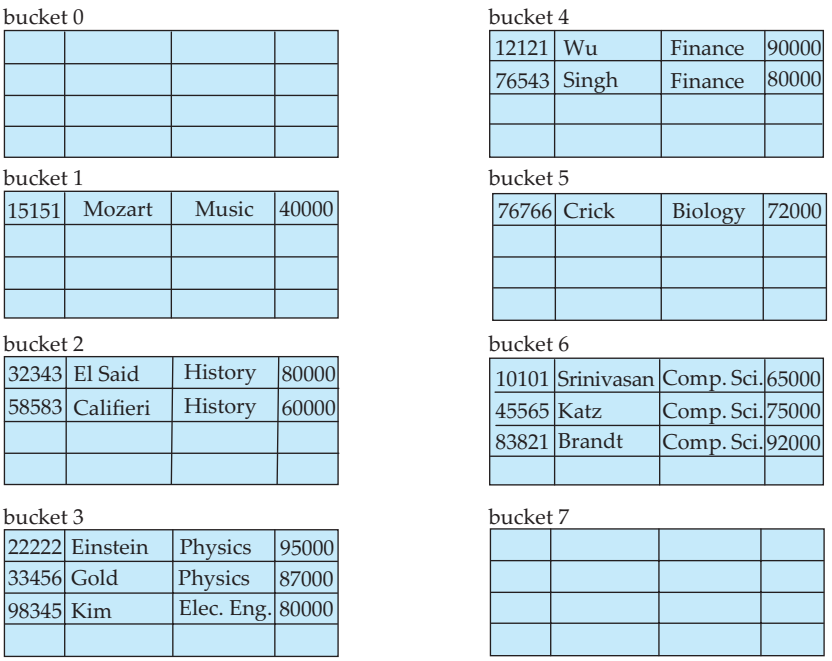



**Figure 11.23** Hash organization of *instructor* file, with *dept_name* as the key.

each bucket has the same number of different *salary* values), but is not random. Records with salaries between $60,001 and $70,000 are far more common than are records with salaries between $30,001 and $40,000. As a result, the distribution of records is not uniform—some buckets receive more records than others do. If the function has a random distribution, even if there are such correlations in the search keys, the randomness of the distribution will make it very likely that all buckets will have roughly the same number of records, as long as each search key occurs in only a small fraction of the records. (If a single search key occurs in a large fraction of the records, the bucket containing it is likely to have more records than other buckets, regardless of the hash function used.)

Typical hash functions perform computation on the internal binary machine representation of characters in the search key. A simple hash function of this type first computes the sum of the binary representations of the characters of a key, then returns the sum modulo the number of buckets.

Figure 11.23 shows the application of such a scheme, with eight buckets, to the *instructor* file, under the assumption that the $i$th letter in the alphabet is represented by the integer $i$.

The following hash function is a better alternative for hashing strings. Let $s$ be a string of length $n$, and let $s[i]$ denote the $i$th byte of the string. The hash function is defined as:

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-1]$$

The function can be implemented efficiently by setting the hash result initially to 0, and iterating from the first to the last character of the string, at each step multiplying the hash value by 31 and then adding the next character (treated as an integer). The above expression would appear to result in a very large number, but it is actually computed with fixed-size positive integers; the result of each multiplication and addition is thus automatically computed modulo the largest possible integer value plus 1. The result of the above function modulo the number of buckets can then be used for indexing.

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

### 11.6.2 Handling of Bucket Overflows

So far, we have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur. Bucket overflow can occur for several reasons:

- **Insufficient buckets**. The number of buckets, which we denote $n_B$, must be chosen such that $n_B > n_r/f_r$, where $n_r$ denotes the total number of records that will be stored and $f_r$ denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.
- **Skew**. Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket **skew**. Skew can occur for two reasons:
  1. Multiple records may have the same search key.
  2. The chosen hash function may result in nonuniform distribution of search keys.

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

Despite allocation of a few more buckets than required, bucket overflow can still occur. We handle bucket overflow by using **overflow buckets**. If a record must be inserted into a bucket $b$, and $b$ is already full, the system provides an overflow bucket for $b$, and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list, as in Figure 11.24. Overflow handling using such a linked list is called **overflow chaining**.

We must change the lookup algorithm slightly to handle overflow chaining. As before, the system uses the hash function on the search key to identify a bucket

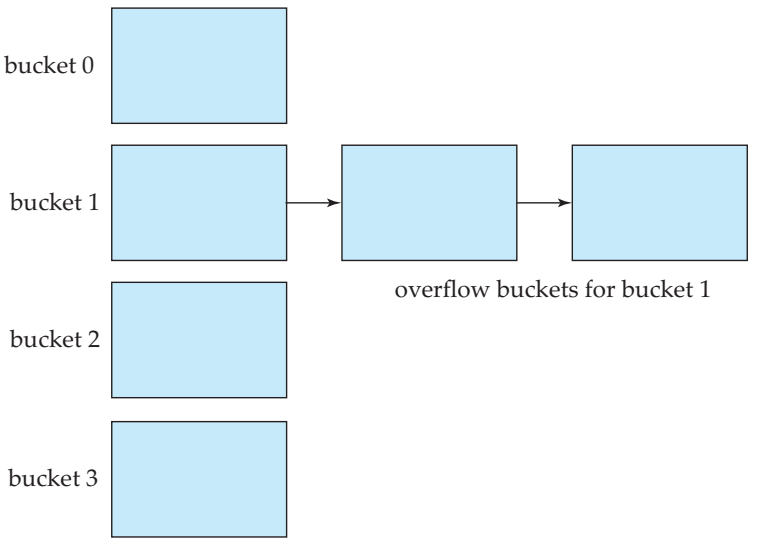



**Figure 11.24** Overflow chaining in a hash structure.

$b$. The system must examine all the records in bucket $b$ to see whether they match the search key, as before. In addition, if bucket $b$ has overflow buckets, the system must examine the records in all the overflow buckets also.

The form of hash structure that we have just described is sometimes referred to as **closed hashing**. Under an alternative approach, called **open hashing**, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets $B$. One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*. Other policies, such as computing further hash functions, are also used. Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations on their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation.

An important drawback to the form of hashing that we have described is that we must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks. Since the function $h$ maps search-key values to a fixed set $B$ of bucket addresses, we waste space if $B$ is made large to handle future growth of the file. If $B$ is too small, the buckets contain records of many different search-key values, and bucket overflows can occur. As the file grows, performance suffers. We study later, in Section 11.7, how the number of buckets and the hash function can be changed dynamically.

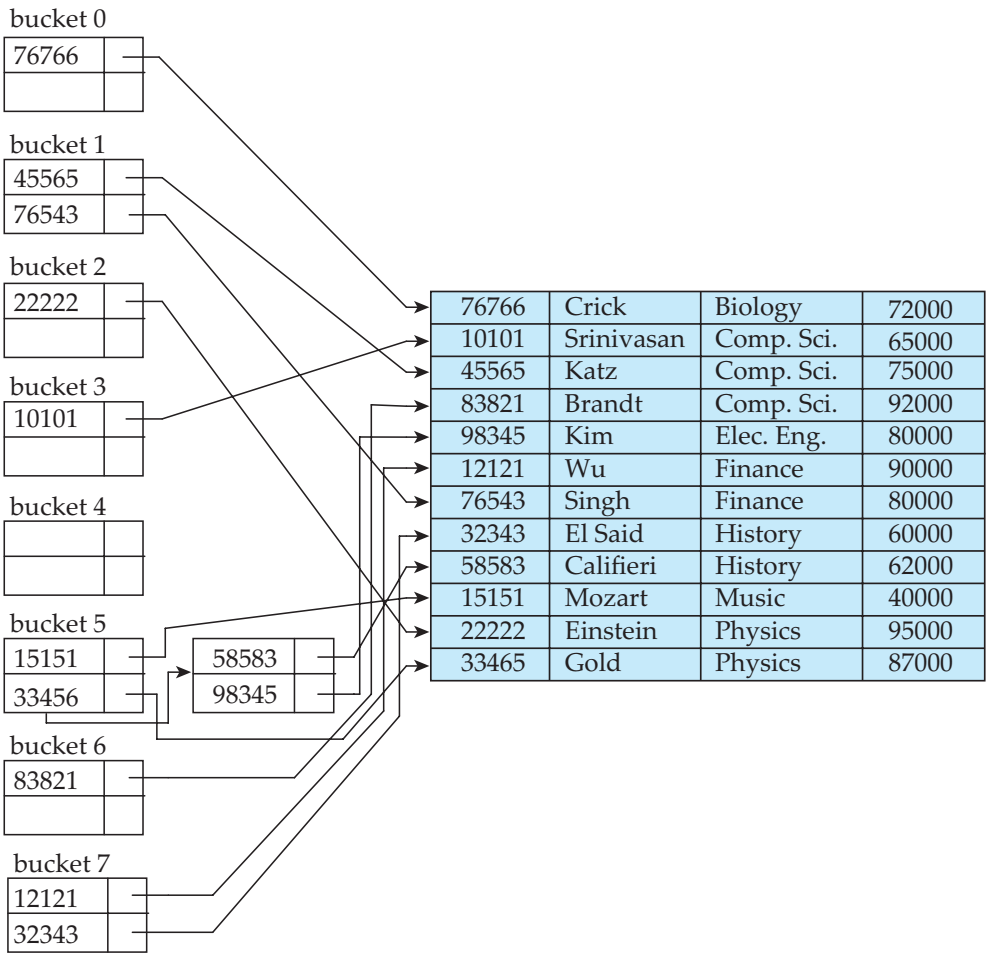



**Figure 11.25** Hash index on search key *ID* of *instructor* file.

### 11.6.3 Hash Indices

Hashing can be used not only for file organization, but also for index-structure creation. A **hash index** organizes the search keys, with their associated pointers, into a hash file structure. We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets). Figure 11.25 shows a secondary hash index on the *instructor* file, for the search key *ID*. The hash function in the figure computes the sum of the digits of the *ID* modulo 8. The hash index has eight buckets, each of size 2 (realistic indices would, of course, have much larger bucket sizes). One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *ID* is a primary key for *instructor*, so each search key has

only one associated pointer. In general, multiple pointers can be associated with each key.

We use the term *hash index* to denote hash file structures as well as secondary hash indices. Strictly speaking, hash indices are only secondary index structures. A hash index is never needed as a clustering index structure, since, if a file itself is organized by hashing, there is no need for a separate hash index structure on it. However, since hash file organization provides the same direct access to records that indexing provides, we pretend that a file organized by hashing also has a clustering hash index on it.

## 11.7 Dynamic Hashing

As we have seen, the need to fix the set $B$ of bucket addresses presents a serious problem with the static hashing technique of the previous section. Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:

1. Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
2. Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
3. Periodically reorganize the hash structure in response to file growth. Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments. This reorganization is a massive, time-consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.

Several **dynamic hashing** techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database. In this section we describe one form of dynamic hashing, called **extendable hashing**. The bibliographical notes provide references to other forms of dynamic hashing.

### 11.7.1 Data Structure

Extendable hashing copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks. As a result, space efficiency is retained. Moreover, since the reorganization is performed on only one bucket at a time, the resulting performance overhead is acceptably low.

With extendable hashing, we choose a hash function $h$ with the desirable properties of uniformity and randomness. However, this hash function generates values over a relatively large range—namely, $b$-bit binary integers. A typical value for $b$ is 32.

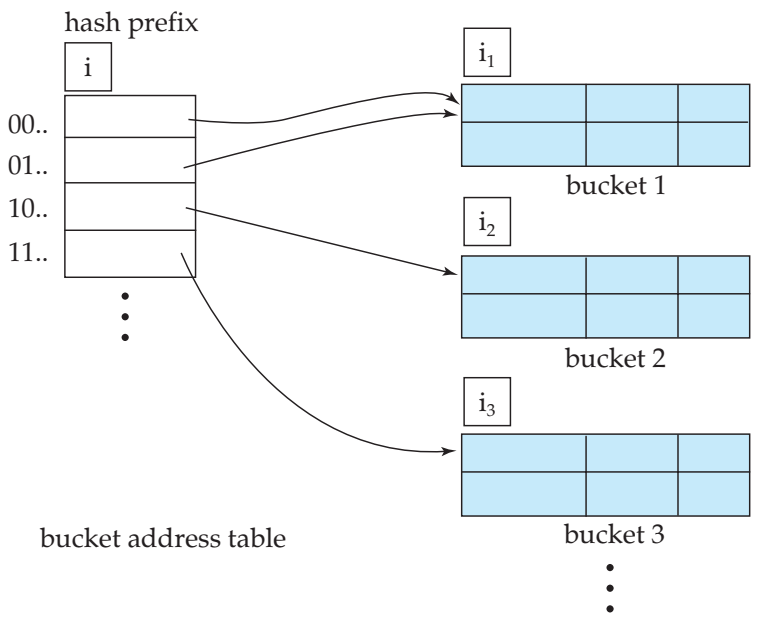



**Figure 11.26** General extendable hash structure.

We do not create a bucket for each hash value. Indeed, $2^{32}$ is over 4 billion, and that many buckets is unreasonable for all but the largest databases. Instead, we create buckets on demand, as records are inserted into the file. We do not use the entire $b$ bits of the hash value initially. At any point, we use $i$ bits, where $0 \leq i \leq b$. These $i$ bits are used as an offset into an additional table of bucket addresses. The value of $i$ grows and shrinks with the size of the database.

Figure 11.26 shows a general extendable hash structure. The $i$ appearing above the bucket address table in the figure indicates that $i$ bits of the hash value $h(K)$ are required to determine the correct bucket for $K$. This number will, of course, change as the file grows. Although $i$ bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket. All such entries will have a common hash prefix, but the length of this prefix may be less than $i$. Therefore, we associate with each bucket an integer giving the length of the common hash prefix. In Figure 11.26 the integer associated with bucket $j$ is shown as $i_j$. The number of bucket-address-table entries that point to bucket $j$ is

$$2^{(i - i_j)}$$

### 11.7.2 Queries and Updates

We now see how to perform lookup, insertion, and deletion on an extendable hash structure.

To locate the bucket containing search-key value $K_l$, the system takes the first $i$ high-order bits of $h(K_l)$, looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry.

To insert a record with search-key value $K_l$, the system follows the same procedure for lookup as before, ending up in some bucket—say, $j$. If there is room in the bucket, the system inserts the record in the bucket. If, on the other hand, the bucket is full, it must split the bucket and redistribute the current records, plus the new one. To split the bucket, the system must first determine from the hash value whether it needs to increase the number of bits that it uses.

- If $i = i_j$, only one entry in the bucket address table points to bucket $j$. Therefore, the system needs to increase the size of the bucket address table so that it can include pointers to the two buckets that result from splitting bucket $j$. It does so by considering an additional bit of the hash value. It increments the value of $i$ by 1, thus doubling the size of the bucket address table. It replaces each entry by two entries, both of which contain the same pointer as the original entry. Now two entries in the bucket address table point to bucket $j$. The system allocates a new bucket (bucket $z$), and sets the second entry to point to the new bucket. It sets $i_j$ and $i_z$ to $i$. Next, it rehashes each record in bucket $j$ and, depending on the first $i$ bits (remember the system has added 1 to $i$), either keeps it in bucket $j$ or allocates it to the newly created bucket.

  The system now reattempts the insertion of the new record. Usually, the attempt will succeed. However, if all the records in bucket $j$, as well as the new record, have the same hash-value prefix, it will be necessary to split a bucket again, since all the records in bucket $j$ and the new record are assigned to the same bucket. If the hash function has been chosen carefully, it is unlikely that a single insertion will require that a bucket be split more than once, unless there are a large number of records with the same search key. If all the records in bucket $j$ have the same search-key value, no amount of splitting will help. In such cases, overflow buckets are used to store the records, as in static hashing.

- If $i > i_j$, then more than one entry in the bucket address table points to bucket $j$. Thus, the system can split bucket $j$ without increasing the size of the bucket address table. Observe that all the entries that point to bucket $j$ correspond to hash prefixes that have the same value on the leftmost $i_j$ bits. The system allocates a new bucket (bucket $z$), and sets $i_j$ and $i_z$ to the value that results from adding 1 to the original $i_j$ value. Next, the system needs to adjust the entries in the bucket address table that previously pointed to bucket $j$. (Note that with the new value for $i_j$, not all the entries correspond to hash prefixes that have the same value on the leftmost $i_j$ bits.) The system leaves the first half of the entries as they were (pointing to bucket $j$), and sets all the remaining entries to point to the newly created bucket (bucket $z$). Next, as in the previous case, the system rehashes each record in bucket $j$, and allocates it either to bucket $j$ or to the newly created bucket $z$.

| *dept_name* | h(*dept_name*) |
|---|---|
| Biology | 0010 1101 1111 1011 0010 1100 0011 0000 |
| Comp. Sci. | 1111 0001 0010 0100 1001 0011 0110 1101 |
| Elec. Eng. | 0100 0011 1010 1100 1100 0110 1101 1111 |
| Finance | 1010 0011 1010 0000 1100 0110 1001 1111 |
| History | 1100 0111 1110 1101 1011 1111 0011 1010 |
| Music | 0011 0101 1010 0110 1100 1001 1110 1011 |
| Physics | 1001 1000 0011 1111 1001 1100 0000 0001 |

**Figure 11.27** Hash function for *dept_name*.

The system then reattempts the insert. In the unlikely case that it again fails, it applies one of the two cases, $i = i_j$ or $i > i_j$, as appropriate.

Note that, in both cases, the system needs to recompute the hash function on only the records in bucket $j$.

To delete a record with search-key value $K_l$, the system follows the same procedure for lookup as before, ending up in some bucket—say, $j$. It removes both the search key from the bucket and the record from the file. The bucket, too, is removed if it becomes empty. Note that, at this point, several buckets can be coalesced, and the size of the bucket address table can be cut in half. The procedure for deciding on which buckets can be coalesced and how to coalesce buckets is left to you to do as an exercise. The conditions under which the bucket address table can be reduced in size are also left to you as an exercise. Unlike coalescing of buckets, changing the size of the bucket address table is a rather expensive operation if the table is large. Therefore it may be worthwhile to reduce the bucket-address-table size only if the number of buckets reduces greatly.

To illustrate the operation of insertion, we use the *instructor* file in Figure 11.1 and assume that the search key is *dept_name* with the 32-bit hash values as appear in Figure 11.27. Assume that, initially, the file is empty, as in Figure 11.28. We insert the records one by one. To illustrate all the features of extendable hashing in a small structure, we shall make the unrealistic assumption that a bucket can hold only two records.

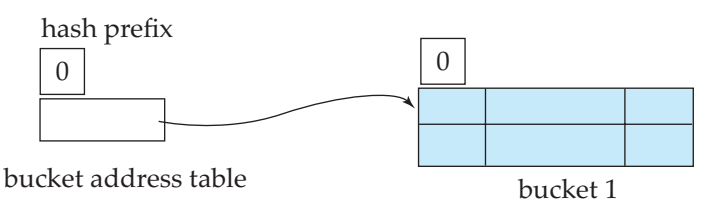



**Figure 11.28** Initial extendable hash structure.

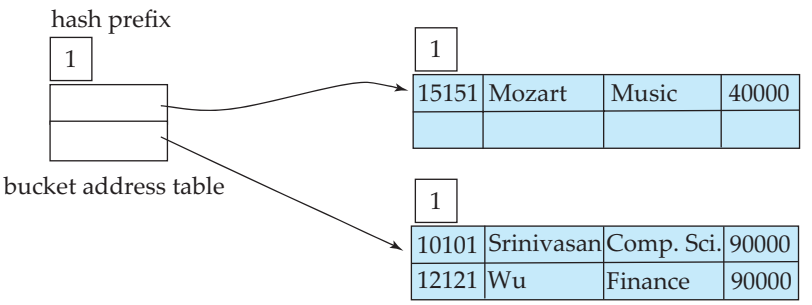



**Figure 11.29** Hash structure after three insertions.

We insert the record (10101, Srinivasan, Comp. Sci., 65000). The bucket address table contains a pointer to the one bucket, and the system inserts the record. Next, we insert the record (12121, Wu, Finance, 90000). The system also places this record in the one bucket of our structure.

When we attempt to insert the next record (15151, Mozart, Music, 40000), we find that the bucket is full. Since $i = i_0$, we need to increase the number of bits that we use from the hash value. We now use 1 bit, allowing us $2^1 = 2$ buckets. This increase in the number of bits necessitates doubling the size of the bucket address table to two entries. The system splits the bucket, placing in the new bucket those records whose search key has a hash value beginning with 1, and leaving in the original bucket the other records. Figure 11.29 shows the state of our structure after the split.

Next, we insert (22222, Einstein, Physics, 95000). Since the first bit of $h$(Physics) is 1, we must insert this record into the bucket pointed to by the "1" entry in the

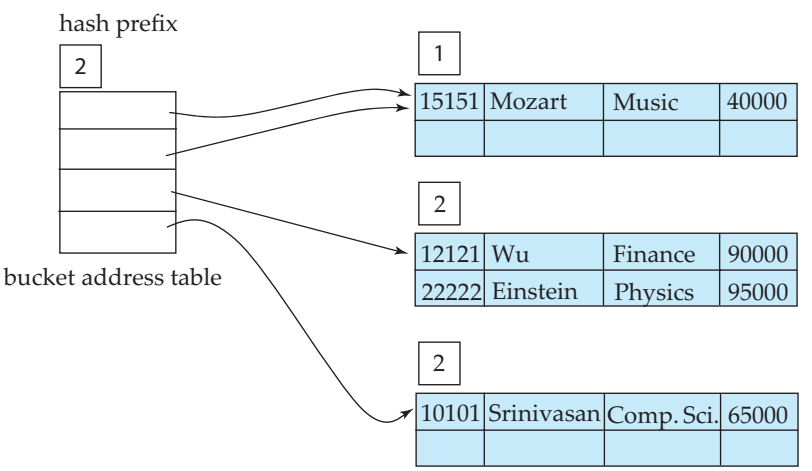



**Figure 11.30** Hash structure after four insertions.

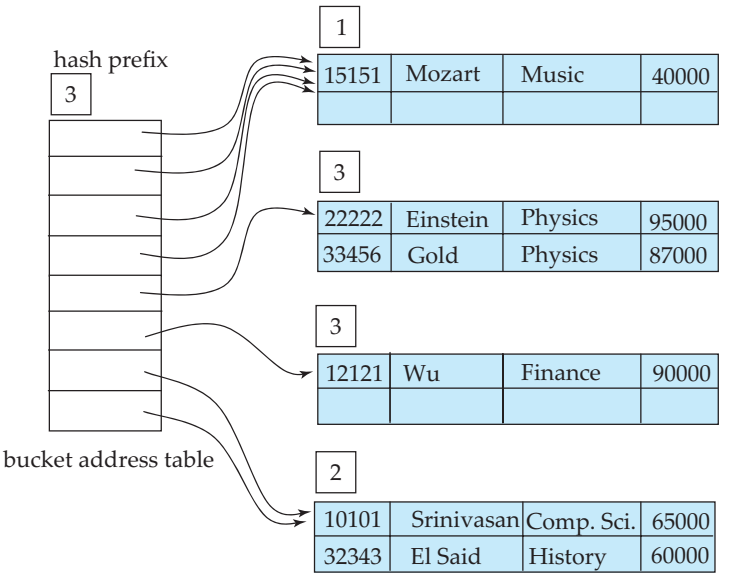



**Figure 11.31** Hash structure after six insertions.

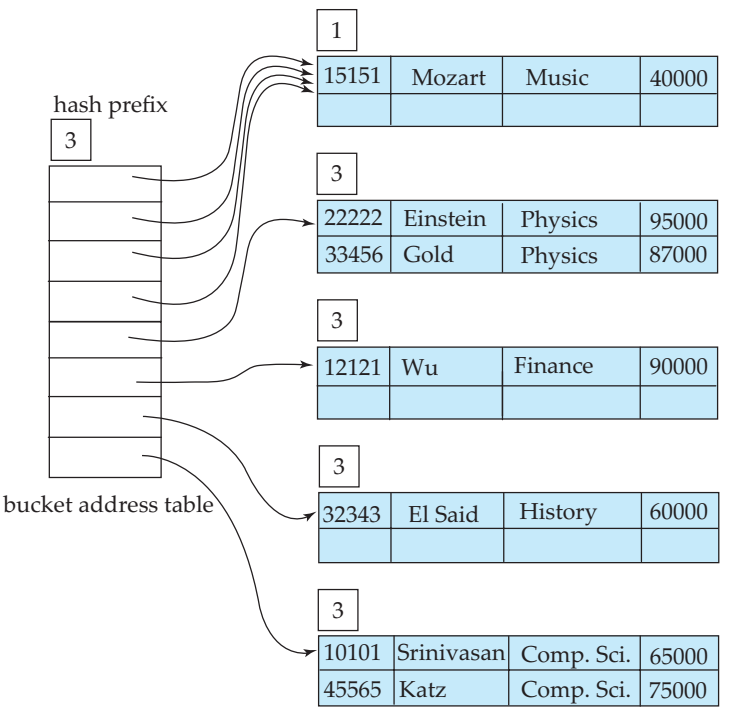



**Figure 11.32** Hash structure after seven insertions.

bucket address table. Once again, we find the bucket full and $i = i_1$. We increase the number of bits that we use from the hash to 2. This increase in the number of bits necessitates doubling the size of the bucket address table to four entries, as in Figure 11.30. Since the bucket of Figure 11.29 for hash prefix 0 was not split, the two entries of the bucket address table of 00 and 01 both point to this bucket.

For each record in the bucket of Figure 11.29 for hash prefix 1 (the bucket being split), the system examines the first 2 bits of the hash value to determine which bucket of the new structure should hold it.

Next, we insert (32343, El Said, History, 60000), which goes in the same bucket as Comp. Sci. The following insertion of (33456, Gold, Physics, 87000) results in a bucket overflow, leading to an increase in the number of bits, and a doubling of the size of the bucket address table (see Figure 11.31).

The insertion of (45565, Katz, Comp. Sci., 75000) leads to another bucket overflow; this overflow, however, can be handled without increasing the number of bits, since the bucket in question has two pointers pointing to it (see Figure 11.32).

Next, we insert the records of "Califieri", "Singh", and "Crick" without any bucket overflow. The insertion of the third Comp. Sci. record (83821, Brandt, Comp. Sci., 92000), however, leads to another overflow. This overflow cannot be handled by increasing the number of bits, since there are three records with exactly the same hash value. Hence the system uses an overflow bucket, as in

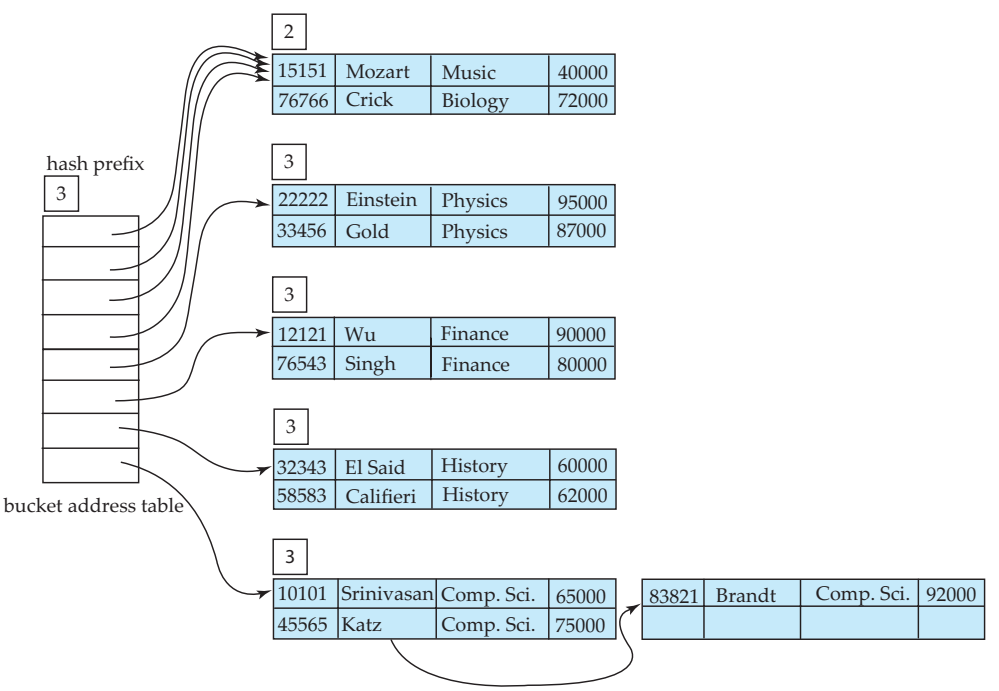



**Figure 11.33** Hash structure after eleven insertions.

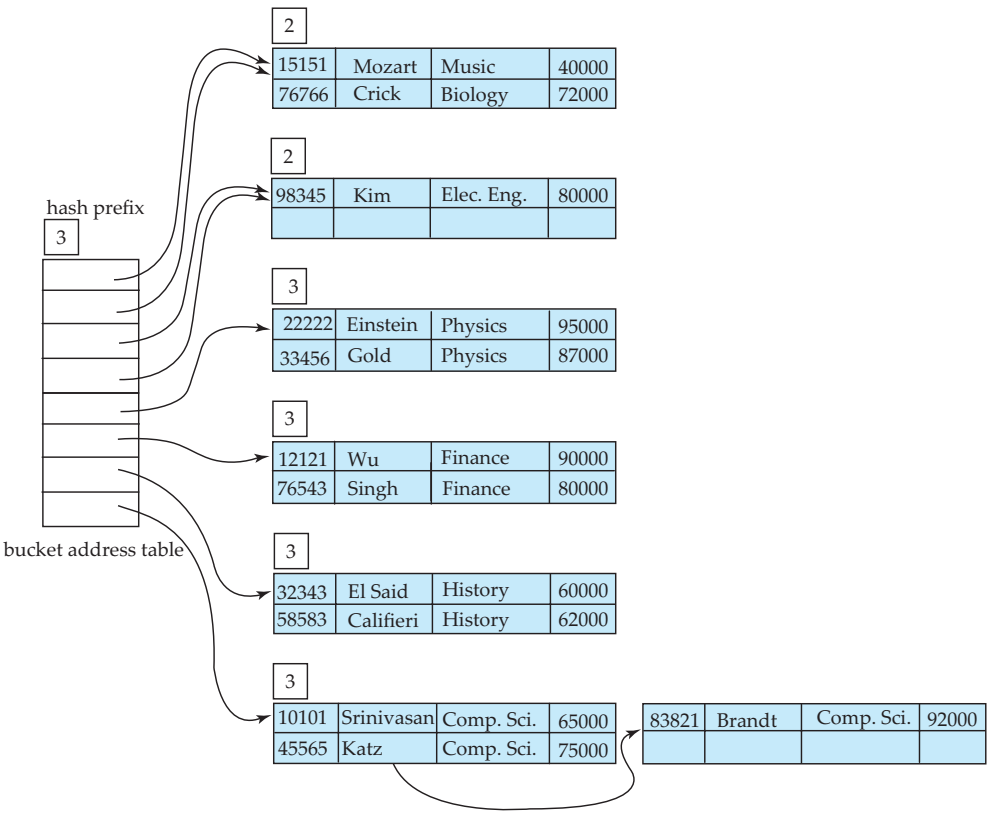



**Figure 11.34** Extendable hash structure for the *instructor* file.

Figure 11.33. We continue in this manner until we have inserted all the *instructor* records of Figure 11.1. The resulting structure appears in Figure 11.34.

### 11.7.3 Static Hashing versus Dynamic Hashing

We now examine the advantages and disadvantages of extendable hashing, compared with static hashing. The main advantage of extendable hashing is that performance does not degrade as the file grows. Furthermore, there is minimal space overhead. Although the bucket address table incurs additional overhead, it contains one pointer for each hash value for the current prefix length. This table is thus small. The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.

A disadvantage of extendable hashing is that lookup involves an additional level of indirection, since the system must access the bucket address table before accessing the bucket itself. This extra reference has only a minor effect on performance. Although the hash structures that we discussed in Section 11.6 do not

have this extra level of indirection, they lose their minor performance advantage as they become full.

Thus, extendable hashing appears to be a highly attractive technique, provided that we are willing to accept the added complexity involved in its implementation. The bibliographical notes reference more detailed descriptions of extendable hashing implementation.

The bibliographical notes also provide references to another form of dynamic hashing called **linear hashing**, which avoids the extra level of indirection associated with extendable hashing, at the possible cost of more overflow buckets.

## 11.8 Comparison of Ordered Indexing and Hashing

We have seen several ordered-indexing schemes and several hashing schemes. We can organize files of records as ordered files by using index-sequential organization or $B^+$-tree organizations. Alternatively, we can organize the files by using hashing. Finally, we can organize them as heap files, where the records are not ordered in any particular way.

Each scheme has advantages in certain situations. A database-system implementor could provide many schemes, leaving the final decision of which schemes to use to the database designer. However, such an approach requires the implementor to write more code, adding both to the cost of the system and to the space that the system occupies. Most database systems support $B^+$-trees and may additionally support some form of hash file organization or hash indices.

To make a choice of file organization and indexing techniques for a relation, the implementor or the database designer must consider the following issues:

- Is the cost of periodic reorganization of the index or hash organization acceptable?
- What is the relative frequency of insertion and deletion?
- Is it desirable to optimize average access time at the expense of increasing the worst-case access time?
- What types of queries are users likely to pose?

We have already examined the first three of these issues, first in our review of the relative merits of specific indexing techniques, and again in our discussion of hashing techniques. The fourth issue, the expected type of query, is critical to the choice of ordered indexing or hashing.

If most queries are of the form:

**select** $A_1, A_2, \ldots, A_n$
**from** $r$
**where** $A_i = c$;

then, to process this query, the system will perform a lookup on an ordered index or a hash structure for attribute $A_i$, for value $c$. For queries of this form, a hashing scheme is preferable. An ordered-index lookup requires time proportional to the log of the number of values in $r$ for $A_i$. In a hash structure, however, the average lookup time is a constant independent of the size of the database. The only advantage to an index over a hash structure for this form of query is that the worst-case lookup time is proportional to the log of the number of values in $r$ for $A_i$. By contrast, for hashing, the worst-case lookup time is proportional to the number of values in $r$ for $A_i$. However, the worst-case lookup time is unlikely to occur with hashing, and hashing is preferable in this case.

Ordered-index techniques are preferable to hashing in cases where the query specifies a range of values. Such a query takes the following form:

**select** $A_1$, $A_2$, ..., $A_n$
**from** $r$
**where** $A_i \leq c_2$ **and** $A_i \geq c_1$;

In other words, the preceding query finds all the records with $A_i$ values between $c_1$ and $c_2$.

Let us consider how we process this query using an ordered index. First, we perform a lookup on value $c_1$. Once we have found the bucket for value $c_1$, we follow the pointer chain in the index to read the next bucket in order, and we continue in this manner until we reach $c_2$.

If, instead of an ordered index, we have a hash structure, we can perform a lookup on $c_1$ and can locate the corresponding bucket—but it is not easy, in general, to determine the next bucket that must be examined. The difficulty arises because a good hash function assigns values randomly to buckets. Thus, there is no simple notion of "next bucket in sorted order." The reason we cannot chain buckets together in sorted order on $A_i$ is that each bucket is assigned many search-key values. Since values are scattered randomly by the hash function, the values in the specified range are likely to be scattered across many or all of the buckets. Therefore, we have to read all the buckets to find the required search keys.

Usually the designer will choose ordered indexing unless it is known in advance that range queries will be infrequent, in which case hashing would be chosen. Hash organizations are particularly useful for temporary files created during query processing, if lookups based on a key value are required, but no range queries will be performed.

## 11.9 Bitmap Indices

Bitmap indices are a specialized type of index designed for easy querying on multiple keys, although each bitmap index is built on a single key.

For bitmap indices to be used, records in a relation must be numbered sequentially, starting from, say, 0. Given a number $n$, it must be easy to retrieve

the record numbered $n$. This is particularly easy to achieve if records are fixed in size, and allocated on consecutive blocks of a file. The record number can then be translated easily into a block number and a number that identifies the record within the block.

Consider a relation $r$, with an attribute $A$ that can take on only one of a small number (for example, 2 to 20) values. For instance, a relation *instructor_info* may have an attribute *gender*, which can take only values m (male) or f (female). Another example would be an attribute *income_level*, where income has been broken up into 5 levels: $L1$: \$0–9999, $L2$: \$10,000–19,999, $L3$: 20,000–39,999, $L4$: 40,000–74,999, and $L5$: 75,000–$\infty$. Here, the raw data can take on many values, but a data analyst has split the values into a small number of ranges to simplify analysis of the data.

### 11.9.1 Bitmap Index Structure

A **bitmap** is simply an array of bits. In its simplest form, a **bitmap index** on the attribute $A$ of relation $r$ consists of one bitmap for each value that $A$ can take. Each bitmap has as many bits as the number of records in the relation. The $i$th bit of the bitmap for value $v_j$ is set to 1 if the record numbered $i$ has the value $v_j$ for attribute $A$. All other bits of the bitmap are set to 0.

In our example, there is one bitmap for the value m and one for f. The $i$th bit of the bitmap for m is set to 1 if the *gender* value of the record numbered $i$ is m. All other bits of the bitmap for m are set to 0. Similarly, the bitmap for f has the value 1 for bits corresponding to records with the value f for the *gender* attribute; all other bits have the value 0. Figure 11.35 shows an example of bitmap indices on a relation *instructor_info*.

We now consider when bitmaps are useful. The simplest way of retrieving all records with value m (or value f) would be to simply read all records of the relation and select those records with value m (or f, respectively). The bitmap index doesn't really help to speed up such a selection. While it would allow us to

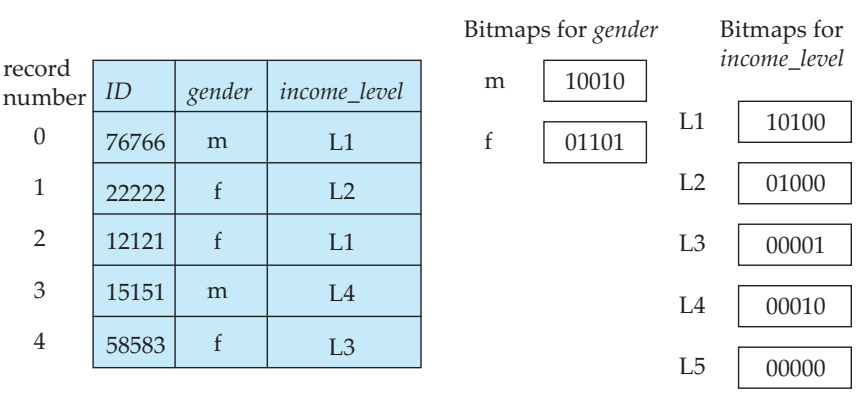



**Figure 11.35** Bitmap indices on relation *instructor_info*.

read only those records for a specific gender, it is likely that every disk block for the file would have to be read anyway.

In fact, bitmap indices are useful for selections mainly when there are selections on multiple keys. Suppose we create a bitmap index on attribute *income_level*, which we described earlier, in addition to the bitmap index on *gender*.

Consider now a query that selects women with income in the range \$10,000 *to* \$19, 999. This query can be expressed as

```
select *
from r
where gender = 'f' and income_level = 'L2';
```

To evaluate this selection, we fetch the bitmaps for *gender* value f and the bitmap for *income_level* value $L2$, and perform an **intersection** (logical-and) of the two bitmaps. In other words, we compute a new bitmap where bit $i$ has value 1 if the $i$th bit of the two bitmaps are both 1, and has a value 0 otherwise. In the example in Figure 11.35, the intersection of the bitmap for $gender = \mathsf{f}$ (01101) and the bitmap for $income\_level = L2$ (01000) gives the bitmap 01000.

Since the first attribute can take two values, and the second can take five values, we would expect only about 1 in 10 records, on an average, to satisfy a combined condition on the two attributes. If there are further conditions, the fraction of records satisfying all the conditions is likely to be quite small. The system can then compute the query result by finding all bits with value 1 in the intersection bitmap and retrieving the corresponding records. If the fraction is large, scanning the entire relation would remain the cheaper alternative.

Another important use of bitmaps is to count the number of tuples satisfying a given selection. Such queries are important for data analysis. For instance, if we wish to find out how many women have an income level $L2$, we compute the intersection of the two bitmaps and then count the number of bits that are 1 in the intersection bitmap. We can thus get the desired result from the bitmap index, without even accessing the relation.

Bitmap indices are generally quite small compared to the actual relation size. Records are typically at least tens of bytes to hundreds of bytes long, whereas a single bit represents the record in a bitmap. Thus the space occupied by a single bitmap is usually less than 1 percent of the space occupied by the relation. For instance, if the record size for a given relation is 100 bytes, then the space occupied by a single bitmap would be $\frac{1}{8}$ of 1 percent of the space occupied by the relation. If an attribute $A$ of the relation can take on only one of eight values, a bitmap index on attribute $A$ would consist of eight bitmaps, which together occupy only 1 percent of the size of the relation.

Deletion of records creates gaps in the sequence of records, since shifting records (or record numbers) to fill gaps would be extremely expensive. To recognize deleted records, we can store an **existence bitmap**, in which bit $i$ is 0 if record $i$ does not exist and 1 otherwise. We shall see the need for existence bitmaps in Section 11.9.2. Insertion of records should not affect the sequence numbering of

other records. Therefore, we can do insertion either by appending records to the end of the file or by replacing deleted records.

### 11.9.2 Efficient Implementation of Bitmap Operations

We can compute the intersection of two bitmaps easily by using a **for** loop: the $i$th iteration of the loop computes the **and** of the $i$th bits of the two bitmaps. We can speed up computation of the intersection greatly by using bit-wise **and** instructions supported by most computer instruction sets. A *word* usually consists of 32 or 64 bits, depending on the architecture of the computer. A bit-wise **and** instruction takes two words as input and outputs a word where each bit is the logical **and** of the bits in corresponding positions of the input words. What is important to note is that a single bit-wise **and** instruction can compute the intersection of 32 or 64 bits *at once*.

If a relation had 1 million records, each bitmap would contain 1 million bits, or equivalently 128 kilobytes. Only 31,250 instructions are needed to compute the intersection of two bitmaps for our relation, assuming a 32-bit word length. Thus, computing bitmap intersections is an extremely fast operation.

Just as bitmap intersection is useful for computing the **and** of two conditions, bitmap union is useful for computing the **or** of two conditions. The procedure for bitmap union is exactly the same as for intersection, except we use bit-wise **or** instructions instead of bit-wise **and** instructions.

The complement operation can be used to compute a predicate involving the negation of a condition, such as **not** (*income-level* = $L1$). The complement of a bitmap is generated by complementing every bit of the bitmap (the complement of 1 is 0 and the complement of 0 is 1). It may appear that **not** (*income_level* = $L1$) can be implemented by just computing the complement of the bitmap for income level $L1$. If some records have been deleted, however, just computing the complement of a bitmap is not sufficient. Bits corresponding to such records would be 0 in the original bitmap, but would become 1 in the complement, although the records don't exist. A similar problem also arises when the value of an attribute is *null*. For instance, if the value of *income_level* is null, the bit would be 0 in the original bitmap for value $L1$, and 1 in the complement bitmap.

To make sure that the bits corresponding to deleted records are set to 0 in the result, the complement bitmap must be intersected with the existence bitmap to turn off the bits for deleted records. Similarly, to handle null values, the complement bitmap must also be intersected with the complement of the bitmap for the value *null*.[2]

Counting the number of bits that are 1 in a bitmap can be done quickly by a clever technique. We can maintain an array with 256 entries, where the $i$th entry stores the number of bits that are 1 in the binary representation of $i$. Set the total count initially to 0. We take each byte of the bitmap, use it to index into this array, and add the stored count to the total count. The number of addition operations is

[2]Handling predicates such as **is unknown** would cause further complications, which would in general require use of an extra bitmap to track which operation results are unknown.

$\frac{1}{8}$ of the number of tuples, and thus the counting process is very efficient. A large array (using $2^{16} = 65{,}536$ entries), indexed by pairs of bytes, would give even higher speedup, but at a higher storage cost.

### 11.9.3 Bitmaps and B$^+$-Trees

Bitmaps can be combined with regular B$^+$-tree indices for relations where a few attribute values are extremely common, and other values also occur, but much less frequently. In a B$^+$-tree index leaf, for each value we would normally maintain a list of all records with that value for the indexed attribute. Each element of the list would be a record identifier, consisting of at least 32 bits, and usually more. For a value that occurs in many records, we store a bitmap instead of a list of records.

Suppose a particular value $v_i$ occurs in $\frac{1}{16}$ of the records of a relation. Let $N$ be the number of records in the relation, and assume that a record has a 64-bit number identifying it. The bitmap needs only 1 bit per record, or $N$ bits in total. In contrast, the list representation requires 64 bits per record where the value occurs, or $64 * N/16 = 4N$ bits. Thus, a bitmap is preferable for representing the list of records for value $v_i$. In our example (with a 64-bit record identifier), if fewer than 1 in 64 records have a particular value, the list representation is preferable for identifying records with that value, since it uses fewer bits than the bitmap representation. If more than 1 in 64 records have that value, the bitmap representation is preferable.

Thus, bitmaps can be used as a compressed storage mechanism at the leaf nodes of B$^+$-trees for those values that occur very frequently.

## 11.10 Index Definition in SQL

The SQL standard does not provide any way for the database user or administrator to control what indices are created and maintained in the database system. Indices are not required for correctness, since they are redundant data structures. However, indices are important for efficient processing of transactions, including both update transactions and queries. Indices are also important for efficient enforcement of integrity constraints.

In principle, a database system can decide automatically what indices to create. However, because of the space cost of indices, as well as the effect of indices on update processing, it is not easy to automatically make the right choices about what indices to maintain. Therefore, most SQL implementations provide the programmer control over creation and removal of indices via data-definition-language commands.

We illustrate the syntax of these commands next. Although the syntax that we show is widely used and supported by many database systems, it is not part of the SQL standard. The SQL standard does not support control of the physical database schema; it restricts itself to the logical database schema.

We create an index with the **create index** command, which takes the form:

**create index** <index-name> **on** <relation-name> (<attribute-list>);

The *attribute-list* is the list of attributes of the relations that form the search key for the index.

To define an index named *dept_index* on the *instructor* relation with *dept_name* as the search key, we write:

**create index** *dept_index* **on** *instructor* (*dept_name*);

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command:

**create unique index** *dept_index* **on** *instructor* (*dept_name*);

declares *dept_name* to be a candidate key for *instructor* (which is probably not what we actually would want for our university database). If, at the time we enter the **create unique index** command, *dept_name* is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index-creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. Note that the **unique** feature is redundant if the database system supports the **unique** declaration of the SQL standard.

Many database systems also provide a way to specify the type of index to be used (such as $B^+$-tree or hashing). Some database systems also permit one of the indices on a relation to be declared to be clustered; the system then stores the relation sorted by the search-key of the clustered index.

The index name we specified for an index is required to drop an index. The **drop index** command takes the form:

**drop index** <index-name>;

## 11.11 Summary

- Many queries reference only a small proportion of the records in a file. To reduce the overhead in searching for these records, we can construct *indices* for the files that store the database.
- Index-sequential files are one of the oldest index schemes used in database systems. To permit fast retrieval of records in search-key order, records are stored sequentially, and out-of-order records are chained together. To allow fast random access, we use an index structure.
- There are two types of indices that we can use: dense indices and sparse indices. Dense indices contain entries for every search-key value, whereas sparse indices contain entries only for some search-key values.

- If the sort order of a search key matches the sort order of a relation, an index on the search key is called a *clustering index*. The other indices are called *nonclustering* or *secondary indices*. Secondary indices improve the performance of queries that use search keys other than the search key of the clustering index. However, they impose an overhead on modification of the database.
- The primary disadvantage of the index-sequential file organization is that performance degrades as the file grows. To overcome this deficiency, we can use a $B^+$*-tree index*.
- A $B^+$-tree index takes the form of a *balanced* tree, in which every path from the root of the tree to a leaf of the tree is of the same length. The height of a $B^+$-tree is proportional to the logarithm to the base $N$ of the number of records in the relation, where each nonleaf node stores $N$ pointers; the value of $N$ is often around 50 or 100. $B^+$-trees are much shorter than other balanced binary-tree structures such as AVL trees, and therefore require fewer disk accesses to locate records.
- Lookup on $B^+$-trees is straightforward and efficient. Insertion and deletion, however, are somewhat more complicated, but still efficient. The number of operations required for lookup, insertion, and deletion on $B^+$-trees is proportional to the logarithm to the base $N$ of the number of records in the relation, where each nonleaf node stores $N$ pointers.
- We can use $B^+$-trees for indexing a file containing records, as well as to organize records into a file.
- B-tree indices are similar to $B^+$-tree indices. The primary advantage of a B-tree is that the B-tree eliminates the redundant storage of search-key values. The major disadvantages are overall complexity and reduced fanout for a given node size. System designers almost universally prefer $B^+$-tree indices over B-tree indices in practice.
- Sequential file organizations require an index structure to locate data. File organizations based on hashing, by contrast, allow us to find the address of a data item directly by computing a function on the search-key value of the desired record. Since we do not know at design time precisely which search-key values will be stored in the file, a good hash function to choose is one that assigns search-key values to buckets such that the distribution is both uniform and random.
- *Static hashing* uses hash functions in which the set of bucket addresses is fixed. Such hash functions cannot easily accommodate databases that grow significantly larger over time. There are several *dynamic hashing techniques* that allow the hash function to be modified. One example is *extendable hashing*, which copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks.

CHAPTER 14

# Transactions

Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user. For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations. Clearly, it is essential that all these operations occur, or that, in case of a failure, none occur. It would be unacceptable if the checking account were debited but the savings account not credited.

Collections of operations that form a single logical unit of work are called **transactions**. A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency. In our funds-transfer example, a transaction computing the customer's total balance might see the checking-account balance before it is debited by the funds-transfer transaction, but see the savings balance after it is credited. As a result, it would obtain an incorrect result.

This chapter introduces the basic concepts of transaction processing. Details on concurrent transaction processing and recovery from failures are in Chapters 15 and 16, respectively. Further topics in transaction processing are discussed in Chapter 26.

## 14.1 Transaction Concept

A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language (for example, C++, or Java), with embedded database accesses in JDBC or ODBC. A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

This collection of steps must appear to the user as a single, indivisible unit. Since a transaction is indivisible, it either executes in its entirety or not at all. Thus, if a transaction begins to execute but fails for whatever reason, any changes to the

database that the transaction may have made must be undone. This requirement holds regardless of whether the transaction itself failed (for example, if it divided by zero), the operating system crashed, or the computer itself stopped operating. As we shall see, ensuring that this requirement is met is difficult since some changes to the database may still be stored only in the main-memory variables of the transaction, while others may have been written to the database and stored on disk. This "all-or-none" property is referred to as **atomicity**.

Furthermore, since a transaction is a single unit, its actions cannot appear to be separated by other database operations not part of the transaction. While we wish to present this user-level impression of transactions, we know that reality is quite different. Even a single SQL statement involves many separate accesses to the database, and a transaction may consist of several SQL statements. Therefore, the database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements. This property is referred to as **isolation**.

Even if the system ensures correct execution of a transaction, this serves little purpose if the system subsequently crashes and, as a result, the system "forgets" about the transaction. Thus, a transaction's actions must persist across crashes. This property is referred to as **durability**.

Because of the above three properties, transactions are an ideal way of structuring interaction with a database. This leads us to impose a requirement on transactions themselves. A transaction must preserve database consistency—if a transaction is run atomically in isolation starting from a consistent database, the database must again be consistent at the end of the transaction. This consistency requirement goes beyond the data integrity constraints we have seen earlier (such as primary-key constraints, referential integrity, **check** constraints, and the like). Rather, transactions are expected to go beyond that to ensure preservation of those application-dependent consistency constraints that are too complex to state using the SQL constructs for data integrity. How this is done is the responsibility of the programmer who codes a transaction. This property is referred to as **consistency**.

To restate the above more concisely, we require that the database system maintain the following properties of the transactions:

- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency**. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$ finished execution before $T_i$ started or $T_j$ started execution after $T_i$ finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

As we shall see later, ensuring the isolation property may have a significant adverse effect on system performance. For this reason, some applications compromise on the isolation property. We shall study these compromises after first studying the strict enforcement of the ACID properties.

## 14.2 A Simple Transaction Model

Because SQL is a powerful and complex language, we begin our study of transactions with a simple database language that focuses on when data are moved from disk to main memory and from main memory to disk. In doing this, we ignore SQL **insert** and **delete** operations, and defer considering them until Section 15.8. The only actual operations on the data are restricted in our simple language to arithmetic operations. Later we shall discuss transactions in a realistic, SQL-based context with a richer set of operations. The data items in our simplified model contain a single data value (a number in our examples). Each data item is identified by a name (typically a single letter in our examples, that is, $A$, $B$, $C$, etc.).

We shall illustrate the transaction concept using a simple bank application consisting of several accounts and a set of transactions that access and update those accounts. Transactions access data using two operations:

- read($X$), which transfers the data item $X$ from the database to a variable, also called $X$, in a buffer in main memory belonging to the transaction that executed the read operation.
- write($X$), which transfers the value in the variable $X$ in the main-memory buffer of the transaction that executed the write to the data item $X$ in the database.

It is important to know if a change to a data item appears only in main memory or if it has been written to the database on disk. In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored elsewhere and executed on the disk later. For now, however, we shall assume that the write operation updates the database immediately. We shall return to this subject in Chapter 16.

Let $T_i$ be a transaction that transfers \$50 from account $A$ to account $B$. This transaction can be defined as:

$T_i$: read($A$);
$A := A - 50$;
write($A$);
read($B$);
$B := B + 50$;
write($B$).

Let us now consider each of the ACID properties. (For ease of presentation, we consider them in an order different from the order A-C-I-D.)

- **Consistency**: The consistency requirement here is that the sum of $A$ and $B$ be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

  Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints, as we discussed in Section 4.4.

- **Atomicity**: Suppose that, just before the execution of transaction $T_i$, the values of accounts $A$ and $B$ are \$1000 and \$2000, respectively. Now suppose that, during the execution of transaction $T_i$, a failure occurs that prevents $T_i$ from completing its execution successfully. Further, suppose that the failure happened after the write($A$) operation but before the write($B$) operation. In this case, the values of accounts $A$ and $B$ reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.

  Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction $T_i$ is executed to completion, there exists a point at which the value of account $A$ is \$950 and the value of account $B$ is \$2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account $A$ is \$950, and the value of account $B$ is \$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

  The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write. This information is written to a file called the *log*. If the transaction does not complete its execution, the database system restores the old values from the log to make it appear as though the transaction never executed. We discuss these ideas further in Section 14.4. Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system**, which we describe in detail in Chapter 16.

- **Durability:** Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of

funds has taken place, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. Protection against loss of data on disk is discussed in Chapter 16. We can guarantee durability by ensuring that either:

1. The updates carried out by the transaction have been written to disk before the transaction completes.
2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

The **recovery system** of the database, described in Chapter 16, is responsible for ensuring durability, in addition to ensuring atomicity.

- **Isolation:** Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

  For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from $A$ to $B$ is executing, with the deducted total written to $A$ and the increased total yet to be written to $B$. If a second concurrently running transaction reads $A$ and $B$ at this intermediate point and computes $A+B$, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on $A$ and $B$ based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

  A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as we shall see in Section 14.5. Other solutions have therefore been developed; they allow multiple transactions to execute concurrently.

  We discuss the problems caused by concurrently executing transactions in Section 14.5. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. We shall discuss the principles of isolation further in Section 14.6. Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control system**, which we discuss later, in Chapter 15.

## 14.3 Storage Structure

To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of how the various data items in the database may be stored and accessed.

In Chapter 10 we saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or nonvolatile storage. We review these terms, and introduce another class of storage, called **stable storage**.

- **Volatile storage**. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.
- **Nonvolatile storage**. Information residing in nonvolatile storage survives system crashes. Examples of nonvolatile storage include secondary storage devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media, and magnetic tapes, used for archival storage. At the current state of technology, nonvolatile storage is slower than volatile storage, particularly for random access. Both secondary and tertiary storage devices, however, are susceptible to failure which may result in loss of information.
- **Stable storage**. Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically *never* cannot be guaranteed—for example, it is possible, although extremely unlikely, that a black hole may envelop the earth and permanently destroy all data!). Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely. To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information. Section 16.2.1 discusses stable-storage implementation.

The distinctions among the various storage types can be less clear in practice than in our presentation. For example, certain systems, for example some RAID controllers, provide battery backup, so that some main memory can survive system crashes and power failures.

For a transaction to be durable, its changes need to be written to stable storage. Similarly, for a transaction to be atomic, log records need to be written to stable storage before any changes are made to the database on disk. Clearly, the degree to which a system ensures durability and atomicity depends on how stable its implementation of stable storage really is. In some cases, a single copy on disk is considered sufficient, but applications whose data are highly valuable and whose

transactions are highly important require multiple copies, or, in other words, a closer approximation of the idealized concept of stable storage.

## 14.4 Transaction Atomicity and Durability

As we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a **log**. Each database modification made by a transaction is first recorded in the log. We record the identifier of the transaction performing the modification, the identifier of the data item being modified, and both the old value (prior to modification) and the new value (after modification) of the data item. Only then is the database itself modified. Maintaining a log provides the possibility of redoing a modification to ensure atomicity and durability as well as the possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution. Details of log-based recovery are discussed in Chapter 16.

A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**. For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account. However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user, and is not handled by the database system. Chapter 26 includes a discussion of compensating transactions.

We need to be more precise about what we mean by *successful completion* of a transaction. We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed**, after successful completion.

The state diagram corresponding to a transaction appears in Figure 14.1. We say that a transaction has committed only if it has entered the committed state.

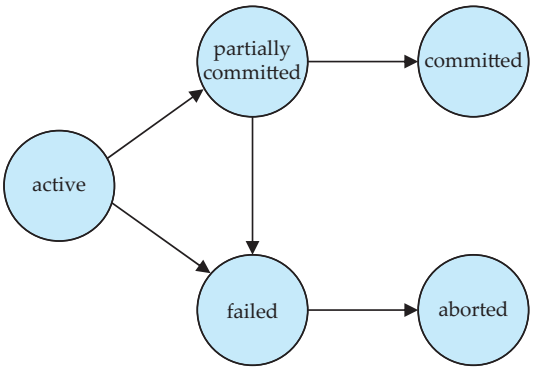



**Figure 14.1** State diagram of a transaction.

Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if it has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

As mentioned earlier, we assume for now that failures do not result in loss of data on disk. Chapter 16 discusses techniques to deal with loss of data on disk.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

We must be cautious when dealing with **observable external writes**, such as writes to a user's screen, or sending email. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system.

Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in a special relation in the database, and to perform the actual writes only after the transaction enters the committed state. If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in nonvolatile storage) when the system is restarted.

Handling external writes can be more complicated in some situations. For example, suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically). It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back in the user's account, needs to be executed when the system is restarted.

As another example, consider a user making a booking over the Web. It is possible that the database system or the application server crashes just after the booking transaction commits. It is also possible that the network connection to the user is lost just after the booking transaction commits. In either case, even though the transaction has committed, the external write has not taken place. To handle such situations, the application must be designed such that when the user connects to the Web application again, she will be able to see whether her transaction had succeeded or not.

For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data unless we are willing to compromise transaction atomicity. In Chapter 26, we discuss alternative transaction models that support long-duration, interactive transactions.

## 14.5 Transaction Isolation

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism

of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

- **Reduced waiting time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system.

When several transactions run concurrently, the isolation property may be violated, resulting in database consistency being destroyed despite the correctness of each individual transaction. In this section, we present the concept of schedules to help identify those executions that are guaranteed to ensure the isolation property and thus database consistency.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called **concurrency-control schemes**. We study concurrency-control schemes in Chapter 15; for now, we focus on the concept of correct concurrent execution.

Consider again the simplified banking system of Section 14.1, which has several accounts, and a set of transactions that access and update those accounts. Let $T_1$ and $T_2$ be two transactions that transfer funds from one account to another. Transaction $T_1$ transfers \$50 from account $A$ to account $B$. It is defined as:

$T_1$: read($A$);
$A := A - 50$;
write($A$);
read($B$);
$B := B + 50$;
write($B$).

Transaction $T_2$ transfers 10 percent of the balance from account $A$ to account $B$. It is defined as:

### TRENDS IN CONCURRENCY

Several current trends in the field of computing are giving rise to an increase in the amount of concurrency possible. As database systems exploit this concurrency to increase overall system performance, there will necessarily be an increasing number of transactions run concurrently.

Early computers had only one processor. Therefore, there was never any real concurrency in the computer. The only concurrency was apparent concurrency created by the operating system as it shared the processor among several distinct tasks or processes. Modern computers are likely to have many processors. These may be truly distinct processors all part of the one computer. However even a single processor may be able to run more than one process at a time by having multiple *cores*. The Intel Core Duo processor is a well-known example of such a multicore processor.

For database systems to take advantage of multiple processors and multiple cores, two approaches are being taken. One is to find parallelism within a single transaction or query. Another is to support a very large number of concurrent transactions.

Many service providers now use large collections of computers rather than large mainframe computers to provide their services. They are making this choice based on the lower cost of this approach. A result of this is yet a further increase in the degree of concurrency that can be supported.

The bibliographic notes refer to texts that describe these advances in computer architecture and parallel computing. Chapter 18 describes algorithms for building parallel database systems, which exploit multiple processors and multiple cores.

$T_2$: read($A$);
*temp* := $A$ * 0.1;
$A := A - temp$;
write($A$);
read($B$);
$B := B + temp$;
write($B$).

Suppose the current values of accounts $A$ and $B$ are \$1000 and \$2000, respectively. Suppose also that the two transactions are executed one at a time in the order $T_1$ followed by $T_2$. This execution sequence appears in Figure 14.2. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of $T_1$ appearing in the left column and instructions of $T_2$ appearing in the right column. The final values of accounts $A$ and $B$, after the execution in Figure 14.2 takes place, are \$855 and \$2145, respectively. Thus, the total amount of money in accounts $A$ and $B$—that is, the sum $A + B$—is preserved after the execution of both transactions.

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

**Figure 14.2** Schedule 1 — a serial schedule in which $T_1$ is followed by $T_2$.

Similarly, if the transactions are executed one at a time in the order $T_2$ followed by $T_1$, then the corresponding execution sequence is that of Figure 14.3. Again, as expected, the sum $A + B$ is preserved, and the final values of accounts $A$ and $B$ are \$850 and \$2150, respectively.

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

**Figure 14.3** Schedule 2 — a serial schedule in which $T_2$ is followed by $T_1$.

The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction. For example, in transaction $T_1$, the instruction write($A$) must appear before the instruction read($B$), in any valid schedule. Note that we include in our schedules the **commit** operation to indicate that the transaction has entered the committed state. In the following discussion, we shall refer to the first execution sequence ($T_1$ followed by $T_2$) as schedule 1, and to the second execution sequence ($T_2$ followed by $T_1$) as schedule 2.

These schedules are **serial**: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Recalling a well-known formula from combinatorics, we note that, for a set of $n$ transactions, there exist $n$ factorial ($n!$) different valid serial schedules.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.[1]

Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure 14.4. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order $T_1$ followed by $T_2$. The sum $A + B$ is indeed preserved.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure 14.5. After the execution of this schedule, we arrive at a state where the final values of accounts $A$ and $B$ are \$950 and \$2100, respectively. This final state is an *inconsistent state*, since we have gained \$50 in the process of the concurrent execution. Indeed, the sum $A + B$ is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The **concurrency-control** component of the database system carries out this task.

[1]The number of possible schedules for a set of $n$ transactions is very large. There are $n!$ different serial schedules. Considering all the possible ways that steps of transactions might be interleaved, the total number of possible schedules is much larger than $n!$.

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

**Figure 14.4** Schedule 3 — a concurrent schedule equivalent to schedule 1.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable** schedules.

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

**Figure 14.5** Schedule 4 — a concurrent schedule resulting in an inconsistent state.

## 14.6 Serializability

Before we can consider how the concurrency-control component of the database system can ensure serializability, we consider how to determine when a schedule is serializable. Certainly, serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable. Since transactions are programs, it is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not consider the various types of operations that a transaction can perform on a data item, but instead consider only two operations: read and write. We assume that, between a read($Q$) instruction and a write($Q$) instruction on a data item $Q$, a transaction may perform an arbitrary sequence of operations on the copy of $Q$ that is residing in the local buffer of the transaction. In this model, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. Commit operations, though relevant, are not considered until Section 14.7. We therefore may show only read and write instructions in schedules, as we do for schedule 3 in Figure 14.6.

In this section, we discuss different forms of schedule equivalence, but focus on a particular form called **conflict serializability**.

Let us consider a schedule $S$ in which there are two consecutive instructions, $I$ and $J$, of transactions $T_i$ and $T_j$, respectively ($i \neq j$). If $I$ and $J$ refer to different data items, then we can swap $I$ and $J$ without affecting the results of any instruction in the schedule. However, if $I$ and $J$ refer to the same data item $Q$, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. $I$ = read($Q$), $J$ = read($Q$). The order of $I$ and $J$ does not matter, since the same value of $Q$ is read by $T_i$ and $T_j$, regardless of the order.
2. $I$ = read($Q$), $J$ = write($Q$). If $I$ comes before $J$, then $T_i$ does not read the value of $Q$ that is written by $T_j$ in instruction $J$. If $J$ comes before $I$, then $T_i$ reads the value of $Q$ that is written by $T_j$. Thus, the order of $I$ and $J$ matters.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| | write($A$) |
| read($B$) | |
| write($B$) | |
| | read($B$) |
| | write($B$) |

**Figure 14.6** Schedule 3 — showing only the read and write instructions.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| read($B$) | |
| | write($A$) |
| write($B$) | |
| | read($B$) |
| | write($B$) |

**Figure 14.7** Schedule 5 — schedule 3 after swapping of a pair of instructions.

3. $I$ = write($Q$), $J$ = read($Q$). The order of $I$ and $J$ matters for reasons similar to those of the previous case.
4. $I$ = write($Q$), $J$ = write($Q$). Since both instructions are write operations, the order of these instructions does not affect either $T_i$ or $T_j$. However, the value obtained by the next read($Q$) instruction of $S$ is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write($Q$) instruction after $I$ and $J$ in $S$, then the order of $I$ and $J$ directly affects the final value of $Q$ in the database state that results from schedule $S$.

Thus, only in the case where both $I$ and $J$ are read instructions does the relative order of their execution not matter.

We say that $I$ and $J$ **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule 3in Figure 14.6. The write($A$) instruction of $T_1$ conflicts with the read($A$) instruction of $T_2$. However, the write($A$) instruction of $T_2$ does not conflict with the read($B$) instruction of $T_1$, because the two instructions access different data items.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| read($B$) | |
| write($B$) | |
| | read($A$) |
| | write($A$) |
| | read($B$) |
| | write($B$) |

**Figure 14.8** Schedule 6 — a serial schedule that is equivalent to schedule 3.

| $T_3$ | $T_4$ |
|---|---|
| read($Q$) | |
| | write($Q$) |
| write($Q$) | |

**Figure 14.9** Schedule 7.

Let $I$ and $J$ be consecutive instructions of a schedule $S$. If $I$ and $J$ are instructions of different transactions and $I$ and $J$ do not conflict, then we can swap the order of $I$ and $J$ to produce a new schedule $S'$. $S$ is equivalent to $S'$, since all instructions appear in the same order in both schedules except for $I$ and $J$, whose order does not matter.

Since the write($A$) instruction of $T_2$ in schedule 3 of Figure 14.6 does not conflict with the read($B$) instruction of $T_1$, we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure 14.7. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

We continue to swap nonconflicting instructions:

- Swap the read($B$) instruction of $T_1$ with the read($A$) instruction of $T_2$.
- Swap the write($B$) instruction of $T_1$ with the write($A$) instruction of $T_2$.
- Swap the write($B$) instruction of $T_1$ with the read($A$) instruction of $T_2$.

The final result of these swaps, schedule 6 of Figure 14.8, is a serial schedule. Note that schedule 6 is exactly the same as schedule 1, but it shows only the read and write instructions. Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.

If a schedule $S$ can be transformed into a schedule $S'$ by a series of swaps of nonconflicting instructions, we say that $S$ and $S'$ are **conflict equivalent**.[2]

Not all serial schedules are conflict equivalent to each other. For example, schedules 1 and 2 are not conflict equivalent.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule $S$ is **conflict serializable** if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Finally, consider schedule 7 of Figure 14.9; it consists of only the significant operations (that is, the read and write) of transactions $T_3$ and $T_4$. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule $<T_3,T_4>$ or the serial schedule $<T_4,T_3>$.

[2]We use the term *conflict equivalent* to distinguish the way we have just defined equivalence from other definitions that we shall discuss later on in this section.


**Figure 14.10** Precedence graph for (a) schedule 1 and (b) schedule 2.

We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule $S$. We construct a directed graph, called a **precedence graph**, from $S$. This graph consists of a pair $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

1. $T_i$ executes write($Q$) before $T_j$ executes read($Q$).
2. $T_i$ executes read($Q$) before $T_j$ executes write($Q$).
3. $T_i$ executes write($Q$) before $T_j$ executes write($Q$).

If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule $S'$ equivalent to $S$, $T_i$ must appear before $T_j$.

For example, the precedence graph for schedule 1 in Figure 14.10a contains the single edge $T_1 \rightarrow T_2$, since all the instructions of $T_1$ are executed before the first instruction of $T_2$ is executed. Similarly, Figure 14.10b shows the precedence graph for schedule 2 with the single edge $T_2 \rightarrow T_1$, since all the instructions of $T_2$ are executed before the first instruction of $T_1$ is executed.

The precedence graph for schedule 4 appears in Figure 14.11. It contains the edge $T_1 \rightarrow T_2$, because $T_1$ executes read($A$) before $T_2$ executes write($A$). It also contains the edge $T_2 \rightarrow T_1$, because $T_2$ executes read($B$) before $T_1$ executes write($B$).

If the precedence graph for $S$ has a cycle, then schedule $S$ is not conflict serializable. If the graph contains no cycles, then the schedule $S$ is conflict serializable.

A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**. There are, in general, several possible linear orders that

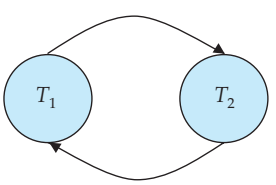



**Figure 14.11** Precedence graph for schedule 4.

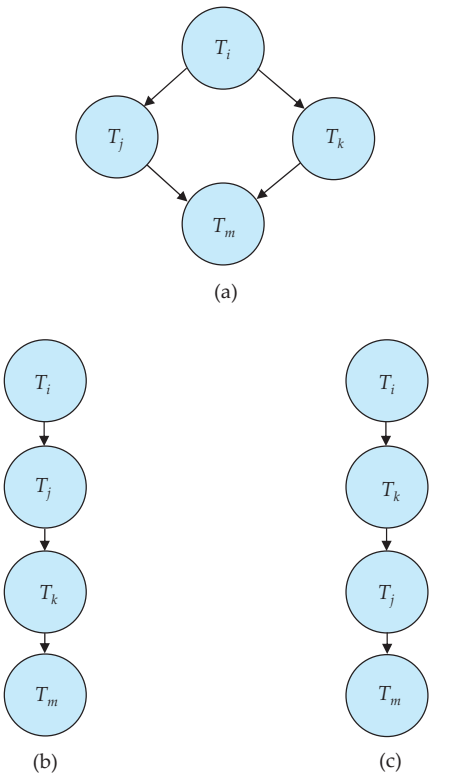



**Figure 14.12** Illustration of topological sorting.

can be obtained through a topological sort. For example, the graph of Figure 14.12a has the two acceptable linear orderings shown in Figures 14.12b and 14.12c.

Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle-detection algorithms can be found in standard textbooks on algorithms. Cycle-detection algorithms, such as those based on depth-first search, require on the order of $n^2$ operations, where $n$ is the number of vertices in the graph (that is, the number of transactions).

Returning to our previous examples, note that the precedence graphs for schedules 1 and 2 (Figure 14.10) indeed do not contain cycles. The precedence graph for schedule 4 (Figure 14.11), on the other hand, contains a cycle, indicating that this schedule is not conflict serializable.

It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent. For example, consider transaction $T_5$, which transfers \$10 from account $B$ to account $A$. Let schedule 8 be as defined in Figure 14.13. We claim that schedule 8 is not conflict equivalent to the serial schedule $<T_1, T_5>$, since, in schedule 8, the write($B$) instruction of $T_5$ conflicts with the read($B$) instruction of $T_1$. This creates an edge $T_5 \rightarrow T_1$ in the precedence graph. Similarly, we see that the write($A$) instruction of $T_1$ conflicts with the read instruction of $T_5$

| $T_1$ | $T_5$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write($A$) | |
| | read($B$) |
| | $B := B - 10$ |
| | write($B$) |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($A$) |
| | $A := A + 10$ |
| | write($A$) |

**Figure 14.13** Schedule 8.

creating an edge $T_1 \rightarrow T_5$. This shows that the precedence graph has a cycle and that schedule 8 is not serializable. However, the final values of accounts $A$ and $B$ after the execution of either schedule 8 or the serial schedule $<T_1,T_5>$ are the same—$960 and $2040, respectively.

We can see from this example that there are less-stringent definitions of schedule equivalence than conflict equivalence. For the system to determine that schedule 8 produces the same outcome as the serial schedule $<T_1,T_5>$, it must analyze the computation performed by $T_1$ and $T_5$, rather than just the read and write operations. In general, such analysis is hard to implement and is computationally expensive. In our example, the final result is the same as that of a serial schedule because of the mathematical fact that addition and subtraction are commutative. While this may be easy to see in our simple example, the general case is not so easy since a transaction may be expressed as a complex SQL statement, a Java program with JDBC calls, etc.

However, there are other definitions of schedule equivalence based purely on the read and write operations. One such definition is *view equivalence*, a definition that leads to the concept of *view serializability*. View serializability is not used in practice due to its high degree of computational complexity.[3] We therefore defer discussion of view serializability to Chapter 15, but, for completeness, note here that the example of schedule 8 is not view serializable.

## 14.7 Transaction Isolation and Atomicity

So far, we have studied schedules while assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

[3]Testing for view serializability has been proven to be NP-complete, which means that it is virtually certain that no efficient test for view serializability exists.

| $T_6$ | $T_7$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| | commit |
| read($B$) | |

**Figure 14.14** Schedule 9, a nonrecoverable schedule.

If a transaction $T_i$ fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, the atomicity property requires that any transaction $T_j$ that is dependent on $T_i$ (that is, $T_j$ has read data written by $T_i$) is also aborted. To achieve this, we need to place restrictions on the type of schedules permitted in the system.

In the following two subsections, we address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure. We describe in Chapter 15 how to ensure that only such acceptable schedules are generated.

### 14.7.1 Recoverable Schedules

Consider the partial schedule 9 in Figure 14.14, in which $T_7$ is a transaction that performs only one instruction: read($A$). We call this a **partial schedule** because we have not included a **commit** or **abort** operation for $T_6$. Notice that $T_7$ commits immediately after executing the read($A$) instruction. Thus, $T_7$ commits while $T_6$ is still in the active state. Now suppose that $T_6$ fails before it commits. $T_7$ has read the value of data item $A$ written by $T_6$. Therefore, we say that $T_7$ is **dependent** on $T_6$. Because of this, we must abort $T_7$ to ensure atomicity. However, $T_7$ has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of $T_6$.

Schedule 9 is an example of a *nonrecoverable* schedule. A **recoverable schedule** is one where, for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads a data item previously written by $T_i$, the commit operation of $T_i$ appears before the commit operation of $T_j$. For the example of schedule 9 to be recoverable, $T_7$ would have to delay committing until after $T_6$ commits.

### 14.7.2 Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction $T_i$, we may have to roll back several transactions. Such situations occur if transactions have read data written by $T_i$. As an illustration, consider the partial schedule of Figure 14.15. Transaction $T_8$ writes a value of $A$ that is read by transaction $T_9$. Transaction $T_9$ writes a value of $A$ that is read by transaction $T_{10}$. Suppose that, at this point, $T_8$ fails. $T_8$ must be rolled back. Since $T_9$ is dependent on $T_8$, $T_9$ must be rolled back. Since $T_{10}$ is dependent on $T_9$, $T_{10}$ must be rolled back. This

| $T_8$ | $T_9$ | $T_{10}$ |
|---|---|---|
| read($A$) | | |
| read($B$) | | |
| write($A$) | | |
| | read($A$) | |
| | write($A$) | |
| | | read($A$) |
| abort | | |

**Figure 14.15** Schedule 10.

phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Formally, a **cascadeless schedule** is one where, for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads a data item previously written by $T_i$, the commit operation of $T_i$ appears before the read operation of $T_j$. It is easy to verify that every cascadeless schedule is also recoverable.

## 14.8 Transaction Isolation Levels

Serializability is a useful concept because it allows programmers to ignore issues related to concurrency when they code transactions. If every transaction has the property that it maintains database consistency if executed alone, then serializability ensures that concurrent executions maintain consistency. However, the protocols required to ensure serializability may allow too little concurrency for certain applications. In these cases, weaker levels of consistency are used. The use of weaker levels of consistency places additional burdens on programmers for ensuring database correctness.

The SQL standard also allows a transaction to specify that it may be executed in such a way that it becomes nonserializable with respect to other transactions. For instance, a transaction may operate at the isolation level of **read uncommitted**, which permits the transaction to read a data item even if it was written by a transaction that has not been committed. SQL provides such features for the benefit of long transactions whose results do not need to be precise. If these transactions were to execute in a serializable fashion, they could interfere with other transactions, causing the others' execution to be delayed.

The isolation levels specified by the SQL standard are as follows:

- **Serializable** usually ensures serializable execution. However, as we shall explain shortly, some database systems implement this isolation level in a manner that may, in certain cases, allow nonserializable executions.

- **Repeatable read** allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it. However, the transaction may not be serializable with respect to other transactions. For instance, when it is searching for data satisfying some conditions, a transaction may find some of the data inserted by a committed transaction, but may not find other data inserted by the same transaction.
- **Read committed** allows only committed data to be read, but does not require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.
- **Read uncommitted** allows uncommitted data to be read. It is the lowest isolation level allowed by SQL.

All the isolation levels above additionally disallow **dirty writes**, that is, they disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted.

Many database systems run, by default, at the read-committed isolation level. In SQL, it is possible to set the isolation level explicitly, rather than accepting the system's default setting. For example, the statement "**set transaction isolation level serializable;**" sets the isolation level to serializable; any of the other isolation levels may be specified instead. The above syntax is supported by Oracle, PostgreSQL and SQL Server; DB2 uses the syntax "**change isolation level**," with its own abbreviations for isolation levels.

Changing of the isolation level must be done as the first statement of a transaction. Further, automatic commit of individual statements must be turned off, if it is on by default; API functions, such as the JDBC method Connection.setAutoCommit(false) which we saw in Section 5.1.1.7, can be used to do so. Further, in JDBC the method Connection.setTransactionIsolation(int level) can be used to set the isolation level; see the JDBC manuals for details.

An application designer may decide to accept a weaker isolation level in order to improve system performance. As we shall see in Section 14.9 and Chapter 15, ensuring serializability may force a transaction to wait for other transactions or, in some cases, to abort because the transaction can no longer be executed as part of a serializable execution. While it may seem shortsighted to risk database consistency for performance, this trade-off makes sense if we can be sure that the inconsistency that may occur is not relevant to the application.

There are many means of implementing isolation levels. As long as the implementation ensures serializability, the designer of a database application or a user of an application does not need to know the details of such implementations, except perhaps for dealing with performance issues. Unfortunately, even if the isolation level is set to **serializable**, some database systems actually implement a weaker level of isolation, which does not rule out every possible nonserializable execution; we revisit this issue in Section 14.9. If weaker levels of isolation are used, either explicitly or implicitly, the application designer has to be aware of some details of the implementation, to avoid or minimize the chance of inconsistency due to lack of serializability.

### SERIALIZABILITY IN THE REAL WORLD

Serializable schedules are the ideal way to ensure consistency, but in our day-to-day lives, we don't impose such stringent requirements. A Web site offering goods for sale may list an item as being in stock, yet by the time a user selects the item and goes through the checkout process, that item might no longer be available. Viewed from a database perspective, this would be a nonrepeatable read.

As another example, consider seat selection for air travel. Assume that a traveler has already booked an itinerary and now is selecting seats for each flight. Many airline Web sites allow the user to step through the various flights and choose a seat, after which the user is asked to confirm the selection. It could be that other travelers are selecting seats or changing their seat selections for the same flights at the same time. The seat availability that the traveler was shown is thus actually changing, but the traveler is shown a snapshot of the seat availability as of when the traveler started the seat selection process.

Even if two travelers are selecting seats at the same time, most likely they will select different seats, and if so there would be no real conflict. However, the transactions are not serializable, since each traveler has read data that was subsequently updated by the other traveler, leading to a cycle in the precedence graph. If two travelers performing seat selection concurrently actually selected the same seat, one of them would not be able to get the seat they selected; however, the situation could be easily resolved by asking the traveler to perform the selection again, with updated seat availability information.

It is possible to enforce serializability by allowing only one traveler to do seat selection for a particular flight at a time. However, doing so could cause significant delays as travelers would have to wait for their flight to become available for seat selection; in particular a traveler who takes a long time to make a choice could cause serious problems for other travelers. Instead, any such transaction is typically broken up into a part that requires user interaction, and a part that runs exclusively on the database. In the example above, the database transaction would check if the seats chosen by the user are still available, and if so update the seat selection in the database. Serializability is ensured only for the transactions that run on the database, without user interaction.

## 14.9 Implementation of Isolation Levels

So far, we have seen what properties a schedule must have if it is to leave the database in a consistent state and allow transaction failures to be handled in a safe manner.

There are various **concurrency-control** policies that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating system time-shares resources (such as CPU time) among the transactions.

As a trivial example of a concurrency-control policy, consider this: A transaction acquires a **lock** on the entire database before it starts and releases the lock after it has committed. While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released. As a result of the locking policy, only one transaction can execute at a time. Therefore, only serial schedules are generated. These are trivially serializable, and it is easy to verify that they are recoverable and cascadeless as well.

A concurrency-control policy such as this one leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start. In other words, it provides a poor degree of concurrency (indeed, no concurrency at all). As we saw in Section 14.5, concurrent execution has substantial performance benefits.

The goal of concurrency-control policies is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, recoverable, and cascadeless.

Here we provide an overview of how some of most important concurrency-control mechanisms work, and we defer the details to Chapter 15.

### 14.9.1 Locking

Instead of locking the entire database, a transaction could, instead, lock only those data items that it accesses. Under such a policy, the transaction must hold locks long enough to ensure serializability, but for a period short enough not to harm performance excessively. Complicating matters are SQL statements like those we saw in Section 14.10, where the data items accessed depend on a **where** clause. In Chapter 15, we present the two-phase locking protocol, a simple, widely used technique that ensures serializability. Stated simply, two-phase locking requires a transaction to have two phases, one where it acquires locks but does not release any, and a second phase where the transaction releases locks but does not acquire any. (In practice, locks are usually released only when the transaction completes its execution and has been either committed or aborted.)

Further improvements to locking result if we have two kinds of locks: shared and exclusive. Shared locks are used for data that the transaction reads and exclusive locks are used for those it writes. Many transactions can hold shared locks on the same data item at the same time, but a transaction is allowed an exclusive lock on a data item only if no other transaction holds any lock (regardless of whether shared or exclusive) on the data item. This use of two modes of locks along with two-phase locking allows concurrent reading of data while still ensuring serializability.

### 14.9.2 Timestamps

Another category of techniques for the implementation of isolation assigns each transaction a **timestamp**, typically when it begins. For each data item, the system keeps two timestamps. The read timestamp of a data item holds the largest (that is, the most recent) timestamp of those transactions that read the data item. The write timestamp of a data item holds the timestamp of the transaction that

wrote the current value of the data item. Timestamps are used to ensure that transactions access each data item in order of the transactions' timestamps if their accesses conflict. When this is not possible, offending transactions are aborted and restarted with a new timestamp.

### 14.9.3 Multiple Versions and Snapshot Isolation

By maintaining more than one version of a data item, it is possible to allow a transaction to read an old version of a data item rather than a newer version written by an uncommitted transaction or by a transaction that should come later in the serialization order. There are a variety of multiversion concurrency-control techniques. One in particular, called **snapshot isolation**, is widely used in practice.

In snapshot isolation, we can imagine that each transaction is given its own version, or snapshot, of the database when it begins.[4] It reads data from this private version and is thus isolated from the updates made by other transactions. If the transaction updates the database, that update appears only in its own version, not in the actual database itself. Information about these updates is saved so that the updates can be applied to the "real" database if the transaction commits.

When a transaction $T$ enters the partially committed state, it then proceeds to the committed state only if no other concurrent transaction has modified data that $T$ intends to update. Transactions that, as a result, cannot commit abort instead.

Snapshot isolation ensures that attempts to read data never need to wait (unlike locking). Read-only transactions cannot be aborted; only those that modify data run a slight risk of aborting. Since each transaction reads its own version or snapshot of the database, reading data does not cause subsequent update attempts by other transactions to wait (unlike locking). Since most transactions are read-only (and most others read more data than they update), this is often a major source of performance improvement as compared to locking.

The problem with snapshot isolation is that, paradoxically, it provides *too much* isolation. Consider two transactions $T$ and $T'$. In a serializable execution, either $T$ sees all the updates made by $T'$ or $T'$ sees all the updates made by $T$, because one must follow the other in the serialization order. Under snapshot isolation, there are cases where neither transaction sees the updates of the other. This is a situation that cannot occur in a serializable execution. In many (indeed, most) cases, the data accesses by the two transactions do not conflict and there is no problem. However, if $T$ reads some data item that $T'$ updates and $T'$ reads some data item that $T$ updates, it is possible that both transactions fail to read the update made by the other. The result, as we shall see in Chapter 15, may be an inconsistent database state that, of course, could not be obtained in any serializable execution.

[4] Of course, in reality, the entire database is not copied. Multiple versions are kept only of those data items that are changed.

Oracle, PostgreSQL, and SQL Server offer the option of snapshot isolation. Oracle and PostgreSQL implement the **serializable** isolation level using snapshot isolation. As a result, their implementation of serializability can, in exceptional circumstances, result in a nonserializable execution being allowed. SQL Server instead includes an additional isolation level beyond the standard ones, called **snapshot**, to offer the option of snapshot isolation.

## 14.10 Transactions as SQL Statements

In Section 4.3, we presented the SQL syntax for specifying the beginning and end of transactions. Now that we have seen some of the issues in ensuring the ACID properties for transactions, we are ready to consider how those properties are ensured when transactions are specified as a sequence of SQL statements rather than the restricted model of simple reads and writes that we considered up to this point.

In our simple model, we assumed a set of data items exists. While our simple model allowed data-item values to be changed, it did not allow data items to be created or deleted. In SQL, however, **insert** statements create new data and **delete** statements delete data. These two statements are, in effect, **write** operations, since they change the database, but their interactions with the actions of other transactions are different from what we saw in our simple model. As an example, consider the following SQL query on our university database that finds all instructors who earn more than $90,000.

**select** *ID*, *name*
**from** *instructor*
**where** *salary* > 90000;

Using our sample *instructor* relation (Appendix A.3), we find that only Einstein and Brandt satisfy the condition. Now assume that around the same time we are running our query, another user inserts a new instructor named "James" whose salary is $100,000.

**insert into** *instructor* **values** ('11111', 'James', 'Marketing', 100000);

The result of our query will be different depending on whether this insert comes before or after our query is run. In a concurrent execution of these transactions, it is intuitively clear that they conflict, but this is a conflict not captured by our simple model. This situation is referred to as the **phantom phenomenon**, because a conflict may exist on "phantom" data.

Our simple model of transactions required that operations operate on a specific data item given as an argument to the operation. In our simple model, we can look at the **read** and **write** steps to see which data items are referenced. But in an SQL statement, the specific data items (tuples) referenced may be determined by a **where** clause predicate. So the same transaction, if run more than once, might

reference different data items each time it is run if the values in the database change between runs.

One way of dealing with the above problem is to recognize that it is not sufficient for concurrency control to consider only the tuples that are accessed by a transaction; the information used to find the tuples that are accessed by the transaction must also be considered for the purpose of concurrency control. The information used to find tuples could be updated by an insertion or deletion, or in the case of an index, even by an update to a search-key attribute. For example, if locking is used for concurrency control, the data structures that track the tuples in a relation, as well as index structures, must be appropriately locked. However, such locking can lead to poor concurrency in some situations; index-locking protocols which maximize concurrency, while ensuring serializability in spite of inserts, deletes, and predicates in queries, are discussed in Section 15.8.3.

Let us consider again the query:

```
select ID, name
from instructor
where salary> 90000;
```

and the following SQL update:

```
update instructor
set salary = salary * 0.9
where name = 'Wu';
```

We now face an interesting situation in determining whether our query conflicts with the update statement. If our query reads the entire *instructor* relation, then it reads the tuple with Wu's data and conflicts with the update. However, if an index were available that allowed our query direct access to those tuples with *salary* > 90000, then our query would not have accessed Wu's data at all because Wu's salary is initially $90,000 in our example instructor relation, and reduces to $81,000 after the update.

However, using the above approach, it would appear that the existence of a conflict depends on a low-level query processing decision by the system that is unrelated to a user-level view of the meaning of the two SQL statements! An alternative approach to concurrency control treats an insert, delete or update as conflicting with a predicate on a relation, if it could affect the set of tuples selected by a predicate. In our example query above, the predicate is "*salary* > 90000", and an update of Wu's salary from $90,000 to a value greater than $90,000, or an update of Einstein's salary from a value greater that $90,000 to a value less than or equal to $90,000, would conflict with this predicate. Locking based on this idea is called **predicate locking**; however predicate locking is expensive, and not used in practice.

## 14.11 Summary

- A *transaction* is a *unit* of program execution that accesses and possibly updates various data items. Understanding the concept of a transaction is critical for understanding and implementing updates of data in a database in such a way that concurrent executions and failures of various forms do not result in the database becoming inconsistent.
- Transactions are required to have the ACID properties: atomicity, consistency, isolation, and durability.
  - Atomicity ensures that either all the effects of a transaction are reflected in the database, or none are; a failure cannot leave the database in a state where a transaction is partially executed.
  - Consistency ensures that, if the database is initially consistent, the execution of the transaction (by itself) leaves the database in a consistent state.
  - Isolation ensures that concurrently executing transactions are isolated from one another, so that each has the impression that no other transaction is executing concurrently with it.
  - Durability ensures that, once a transaction has been committed, that transaction's updates do not get lost, even if there is a system failure.
- Concurrent execution of transactions improves throughput of transactions and system utilization, and also reduces waiting time of transactions.
- The various types of storage in a computer are volatile storage, nonvolatile storage, and stable storage. Data in volatile storage, such as in RAM, are lost when the computer crashes. Data in nonvolatile storage, such as disk, are not lost when the computer crashes, but may occasionally be lost because of failures such as disk crashes. Data in stable storage are never lost.
- Stable storage that must be accessible online is approximated with mirrored disks, or other forms of RAID, which provide redundant data storage. Offline, or archival, stable storage may consist of multiple tape copies of data stored in physically secure locations.
- When several transactions execute concurrently on the database, the consistency of data may no longer be preserved. It is therefore necessary for the system to control the interaction among the concurrent transactions.
  - Since a transaction is a unit that preserves consistency, a serial execution of transactions guarantees that consistency is preserved.
  - A *schedule* captures the key actions of transactions that affect concurrent execution, such as read and write operations, while abstracting away internal details of the execution of the transaction.

  - We require that any schedule produced by concurrent processing of a set of transactions will have an effect equivalent to a schedule produced when these transactions are run serially in some order.
  - A system that guarantees this property is said to ensure *serializability*.
  - There are several different notions of equivalence leading to the concepts of *conflict serializability* and *view serializability*.
- Serializability of schedules generated by concurrently executing transactions can be ensured through one of a variety of mechanisms called *concurrency-control* policies.
- We can test a given schedule for conflict serializability by constructing a *precedence graph* for the schedule, and by searching for absence of cycles in the graph. However, there are more efficient concurrency-control policies for ensuring serializability.
- Schedules must be recoverable, to make sure that if transaction $a$ sees the effects of transaction $b$, and $b$ then aborts, then $a$ also gets aborted.
- Schedules should preferably be cascadeless, so that the abort of a transaction does not result in cascading aborts of other transactions. Cascadelessness is ensured by allowing transactions to only read committed data.
- The concurrency-control–management component of the database is responsible for handling the concurrency-control policies. Chapter 15 describes concurrency-control policies.

## Review Terms

- Transaction
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Inconsistent state
- Storage types
  - Volatile storage
  - Nonvolatile storage
  - Stable storage
- Concurrency control system
- Recovery system
- Transaction state
  - Active
  - Partially committed
  - Failed
  - Aborted
  - Committed
  - Terminated
- Transaction
  - Restart
  - Kill

- Observable external writes
- Concurrent executions
- Serial execution
- Schedules
- Conflict of operations
- Conflict equivalence
- Conflict serializability
- Serializability testing
- Precedence graph
- Serializability order
- Recoverable schedules
- Cascading rollback
- Cascadeless schedules
- Concurrency-control
- Locking
- Multiple versions
- Snapshot isolation

## Practice Exercises

**14.1** Suppose that there is a database system that never fails. Is a recovery manager required for this system?

**14.2** Consider a file system such as the one on your favorite operating system.

a. What are the steps involved in creation and deletion of files, and in writing data to a file?

b. Explain how the issues of atomicity and durability are relevant to the creation and deletion of files and to writing data to files.

**14.3** Database-system implementers have paid much more attention to the ACID properties than have file-system implementers. Why might this be the case?

**14.4** Justify the following statement: Concurrent execution of transactions is more important when data must be fetched from (slow) disk or when transactions are long, and is less important when data are in memory and transactions are very short.

**14.5** Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

**14.6** Consider the precedence graph of Figure 14.16. Is the corresponding schedule conflict serializable? Explain your answer.

**14.7** What is a cascadeless schedule? Why is cascadelessness of schedules desirable? Are there any circumstances under which it would be desirable to allow noncascadeless schedules? Explain your answer.

**14.8** The **lost update** anomaly is said to occur if a transaction $T_j$ reads a data item, then another transaction $T_k$ writes the data item (possibly based on a previous read), after which $T_j$ writes the data item. The update performed by $T_k$ has been lost, since the update done by $T_j$ ignored the value written by $T_k$.

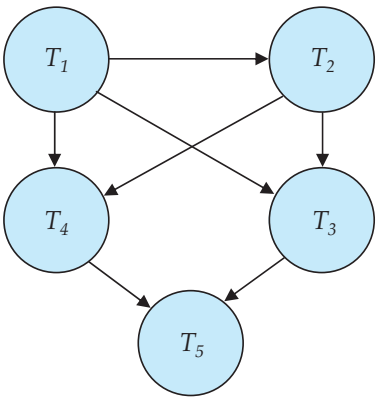



**Figure 14.16** Precedence graph for Practice Exercise 14.6.

a. Give an example of a schedule showing the lost update anomaly.

b. Give an example schedule to show that the lost update anomaly is possible with the **read committed** isolation level.

c. Explain why the lost update anomaly is not possible with the **repeatable read** isolation level.

**14.9** Consider a database for a bank where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs that would present a problem for the bank.

**14.10** Consider a database for an airline where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs, but the airline may be willing to accept it in order to gain better overall performance.

**14.11** The definition of a schedule assumes that operations can be totally ordered by time. Consider a database system that runs on a system with multiple processors, where it is not always possible to establish an exact ordering between operations that executed on different processors. However, operations on a data item can be totally ordered.

Does the above situation cause any problem for the definition of conflict serializability? Explain your answer.

## Exercises

**14.12** List the ACID properties. Explain the usefulness of each.

**14.13** During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through

which a transaction may pass. Explain why each state transition may occur.

**14.14** Explain the distinction between the terms *serial schedule* and *serializable schedule*.

**14.15** Consider the following two transactions:

$T_{13}$: read($A$);
read($B$);
**if** $A = 0$ **then** $B := B + 1$;
write($B$).

$T_{14}$: read($B$);
read($A$);
**if** $B = 0$ **then** $A := A + 1$;
write($A$).

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

b. Show a concurrent execution of $T_{13}$ and $T_{14}$ that produces a nonserializable schedule.

c. Is there a concurrent execution of $T_{13}$ and $T_{14}$ that produces a serializable schedule?

**14.16** Give an example of a serializable schedule with two transactions such that the order in which the transactions commit is different from the serialization order.

**14.17** What is a recoverable schedule? Why is recoverability of schedules desirable? Are there any circumstances under which it would be desirable to allow nonrecoverable schedules? Explain your answer.

**14.18** Why do database systems support concurrent execution of transactions, in spite of the extra programming effort needed to ensure that concurrent execution does not cause any problems?

**14.19** Explain why the read-committed isolation level ensures that schedules are cascade-free.

**14.20** For each of the following isolation levels, give an example of a schedule that respects the specified level of isolation, but is not serializable:

a. Read uncommitted

b. Read committed

c. Repeatable read

**14.21** Suppose that in addition to the operations read and write, we allow an operation pred_read($r$, $P$), which reads all tuples in relation $r$ that satisfy predicate $P$.

a. Give an example of a schedule using the pred_read operation that exhibits the phantom phenomenon, and is nonserializable as a result.

b. Give an example of a schedule where one transaction uses the pred_read operation on relation $r$ and another concurrent transactions deletes a tuple from $r$, but the schedule does not exhibit a phantom conflict. (To do so, you have to give the schema of relation $r$, and show the attribute values of the deleted tuple.)

## Bibliographical Notes

Gray and Reuter [1993] provides detailed textbook coverage of transaction-processing concepts, techniques and implementation details, including concurrency control and recovery issues. Bernstein and Newcomer [1997] provides textbook coverage of various aspects of transaction processing.

The concept of serializability was formalized by Eswaran et al. [1976] in connection to work on concurrency control for System R.

References covering specific aspects of transaction processing, such as concurrency control and recovery, are cited in Chapters 15, 16, and 26.

CHAPTER 15

# Concurrency Control

We saw in Chapter 14 that one of the fundamental properties of a transaction is isolation. When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes. In Chapter 26, we discuss concurrency-control schemes that admit nonserializable schedules. In this chapter, we consider the management of concurrently executing transactions, and we ignore failures. In Chapter 16, we shall see how the system can recover from failures.

As we shall see, there are a variety of concurrency-control schemes. No one scheme is clearly the best; each one has advantages. In practice, the most frequently used schemes are *two-phase locking* and *snapshot isolation*.

## 15.1 Lock-Based Protocols

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item. We introduced the concept of locking in Section 14.9.

### 15.1.1 Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. **Shared**. If a transaction $T_i$ has obtained a **shared-mode lock** (denoted by S) on item $Q$, then $T_i$ can read, but cannot write, $Q$.
2. **Exclusive**. If a transaction $T_i$ has obtained an **exclusive-mode lock** (denoted by X) on item $Q$, then $T_i$ can both read and write $Q$.

| | S | X |
|---|---|---|
| S | true | false |
| X | false | false |

**Figure 15.1** Lock-compatibility matrix comp.

We require that every transaction **request** a lock in an appropriate mode on data item $Q$, depending on the types of operations that it will perform on $Q$. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction. The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

To state this more generally, given a set of lock modes, we can define a **compatibility function** on them as follows: Let $A$ and $B$ represent arbitrary lock modes. Suppose that a transaction $T_i$ requests a lock of mode $A$ on item $Q$ on which transaction $T_j$ ($T_i \neq T_j$) currently holds a lock of mode $B$. If transaction $T_i$ can be granted a lock on $Q$ immediately, in spite of the presence of the mode $B$ lock, then we say mode $A$ is **compatible** with mode $B$. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure 15.1. An element comp($A$, $B$) of the matrix has the value *true* if and only if mode $A$ is compatible with mode $B$.

Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

A transaction requests a shared lock on data item $Q$ by executing the lock-S($Q$) instruction. Similarly, a transaction requests an exclusive lock through the lock-X($Q$) instruction. A transaction can unlock a data item $Q$ by the unlock($Q$) instruction.

To access a data item, transaction $T_i$ must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency-control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, $T_i$ is made to **wait** until all incompatible locks held by other transactions have been released.

Transaction $T_i$ may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

As an illustration, consider again the banking example that we introduced in Chapter 14. Let $A$ and $B$ be two accounts that are accessed by transactions $T_1$

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

**Figure 15.2** Transaction $T_1$.

and $T_2$. Transaction $T_1$ transfers \$50 from account $B$ to account $A$ (Figure 15.2). Transaction $T_2$ displays the total amount of money in accounts $A$ and $B$—that is, the sum $A + B$ (Figure 15.3).

Suppose that the values of accounts $A$ and $B$ are \$100 and \$200, respectively. If these two transactions are executed serially, either in the order $T_1$, $T_2$ or the order $T_2$, $T_1$, then transaction $T_2$ will display the value \$300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 15.4, is possible. In this case, transaction $T_2$ displays \$250, which is incorrect. The reason for this mistake is that the transaction $T_1$ unlocked data item $B$ too early, as a result of which $T_2$ saw an inconsistent state.

The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks. The transaction making a lock request cannot execute its next action until the concurrency-control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction. Exactly when within this interval the lock is granted is not important; we can safely assume that the lock is granted just before the following action of the transaction. We shall therefore drop the column depicting the actions of the concurrency-control manager from all schedules depicted in the rest of the chapter. We let you infer when locks are granted.

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

**Figure 15.3** Transaction $T_2$.

| $T_1$ | $T_2$ | concurreny-control manager |
|---|---|---|
| lock-X($B$) | | |
| | | grant-X($B$, $T_1$) |
| read($B$) | | |
| $B := B - 50$ | | |
| write($B$) | | |
| unlock($B$) | | |
| | lock-S($A$) | |
| | | grant-S($A$, $T_2$) |
| | read($A$) | |
| | unlock($A$) | |
| | lock-S($B$) | |
| | | grant-S($B$, $T_2$) |
| | read($B$) | |
| | unlock($B$) | |
| | display($A + B$) | |
| lock-X($A$) | | |
| | | grant-X($A$, $T_1$) |
| read($A$) | | |
| $A := A - 50$ | | |
| write($A$) | | |
| unlock($A$) | | |

**Figure 15.4** Schedule 1.

Suppose now that unlocking is delayed to the end of the transaction. Transaction $T_3$ corresponds to $T_1$ with unlocking delayed (Figure 15.5). Transaction $T_4$ corresponds to $T_2$ with unlocking delayed (Figure 15.6).

You should verify that the sequence of reads and writes in schedule 1, which lead to an incorrect total of \$250 being displayed, is no longer possible with $T_3$

$T_3$: lock-X($B$);
read($B$);
$B := B - 50$;
write($B$);
lock-X($A$);
read($A$);
$A := A + 50$;
write($A$);
unlock($B$);
unlock($A$).

**Figure 15.5** Transaction $T_3$ (transaction $T_1$ with unlocking delayed).

$T_4$: lock-S($A$);
read($A$);
lock-S($B$);
read($B$);
display($A + B$);
unlock($A$);
unlock($B$).

**Figure 15.6** Transaction $T_4$ (transaction $T_2$ with unlocking delayed).

and $T_4$. Other schedules are possible. $T_4$ will not print out an inconsistent result in any of them; we shall see why later.

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 15.7 for $T_3$ and $T_4$. Since $T_3$ is holding an exclusive-mode lock on $B$ and $T_4$ is requesting a shared-mode lock on $B$, $T_4$ is waiting for $T_3$ to unlock $B$. Similarly, since $T_4$ is holding a shared-mode lock on $A$ and $T_3$ is requesting an exclusive-mode lock on $A$, $T_3$ is waiting for $T_4$ to unlock $A$. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**. When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution. We shall return to the issue of deadlock handling in Section 15.2.

If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur. There are ways to avoid deadlock in some situations, as we shall see in Section 15.1.5. However, in general, deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely

| $T_3$ | $T_4$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | lock-S($B$) |
| lock-X($A$) | |

**Figure 15.7** Schedule 2.

preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules. We shall present several locking protocols that allow only conflict-serializable schedules, and thereby ensure isolation. Before doing so, we introduce some terminology.

Let $\{T_0, T_1, \ldots, T_n\}$ be a set of transactions participating in a schedule $S$. We say that $T_i$ **precedes** $T_j$ in $S$, written $T_i \rightarrow T_j$, if there exists a data item $Q$ such that $T_i$ has held lock mode $A$ on $Q$, and $T_j$ has held lock mode $B$ on $Q$ later, and comp$(A,B)$ = false. If $T_i \rightarrow T_j$, then that precedence implies that in any equivalent serial schedule, $T_i$ must appear before $T_j$. Observe that this graph is similar to the precedence graph that we used in Section 14.6 to test for conflict serializability. Conflicts between instructions correspond to noncompatibility of lock modes.

We say that a schedule $S$ is **legal** under a given locking protocol if $S$ is a possible schedule for a set of transactions that follows the rules of the locking protocol. We say that a locking protocol **ensures** conflict serializability if and only if all legal schedules are conflict serializable; in other words, for all legal schedules the associated $\rightarrow$ relation is acyclic.

### 15.1.2 Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction $T_2$ has a shared-mode lock on a data item, and another transaction $T_1$ requests an exclusive-mode lock on the data item. Clearly, $T_1$ has to wait for $T_2$ to release the shared-mode lock. Meanwhile, a transaction $T_3$ may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to $T_2$, so $T_3$ may be granted the shared-mode lock. At this point $T_2$ may release the lock, but still $T_1$ has to wait for $T_3$ to finish. But again, there may be a new transaction $T_4$ that requests a shared-mode lock on the same data item, and is granted the lock before $T_3$ releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but $T_1$ never gets the exclusive-mode lock on the data item. The transaction $T_1$ may never make progress, and is said to be **starved**.

We can avoid starvation of transactions by granting locks in the following manner: When a transaction $T_i$ requests a lock on a data item $Q$ in a particular mode $M$, the concurrency-control manager grants the lock provided that:

1. There is no other transaction holding a lock on $Q$ in a mode that conflicts with $M$.

2. There is no other transaction that is waiting for a lock on $Q$ and that made its lock request before $T_i$.

Thus, a lock request will never get blocked by a lock request that is made later.

### 15.1.3 The Two-Phase Locking Protocol

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase**. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

For example, transactions $T_3$ and $T_4$ are two phase. On the other hand, transactions $T_1$ and $T_2$ are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction $T_3$, we could move the unlock($B$) instruction to just after the lock-X($A$) instruction, and still retain the two-phase locking property.

We can show that the two-phase locking protocol ensures conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction. Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions. We leave the proof as an exercise for you to do (see Practice Exercise 15.1).

Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions $T_3$ and $T_4$ are two phase, but, in schedule 2 (Figure 15.7), they are deadlocked.

Recall from Section 14.7.2 that, in addition to being serializable, schedules should be cascadeless. Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of Figure 15.8. Each transaction observes the two-phase locking protocol, but the failure of $T_5$ after the read(A) step of $T_7$ leads to cascading rollback of $T_6$ and $T_7$.

Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits.

| $T_5$ | $T_6$ | $T_7$ |
|---|---|---|
| lock-X($A$) | | |
| read($A$) | | |
| lock-S($B$) | | |
| read($B$) | | |
| write($A$) | | |
| unlock($A$) | | |
| | lock-X($A$) | |
| | read($A$) | |
| | write($A$) | |
| | unlock($A$) | |
| | | lock-S($A$) |
| | | read($A$) |

**Figure 15.8** Partial schedule under two-phase locking.

We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit.

Consider the following two transactions, for which we have shown only some of the significant read and write operations:

$T_8$: read($a_1$);
read($a_2$);
...
read($a_n$);
write($a_1$).

$T_9$: read($a_1$);
read($a_2$);
display($a_1 + a_2$).

If we employ the two-phase locking protocol, then $T_8$ must lock $a_1$ in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that $T_8$ needs an exclusive lock on $a_1$ only at the end of its execution, when it writes $a_1$. Thus, if $T_8$ could initially lock $a_1$ in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since $T_8$ and $T_9$ could access $a_1$ and $a_2$ simultaneously.

This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

| $T_8$ | $T_9$ |
|---|---|
| lock-S($a_1$) | |
| | lock-S($a_1$) |
| lock-S($a_2$) | |
| | lock-S($a_2$) |
| lock-S($a_3$) | |
| lock-S($a_4$) | |
| | unlock($a_1$) |
| | unlock($a_2$) |
| lock-S($a_n$) | |
| upgrade($a_1$) | |

**Figure 15.9** Incomplete schedule with a lock conversion.

Returning to our example, transactions $T_8$ and $T_9$ can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure 15.9, where only some of the locking instructions are shown.

Note that a transaction attempting to upgrade a lock on an item $Q$ may be forced to wait. This enforced wait occurs if $Q$ is currently locked by *another* transaction in shared mode.

Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict-serializable schedules through non-two-phase locking protocols, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data items in the database. We shall see examples when we consider other locking protocols later in this chapter.

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction $T_i$ issues a read($Q$) operation, the system issues a lock-S($Q$) instruction followed by the read($Q$) instruction.
- When $T_i$ issues a write($Q$) operation, the system checks to see whether $T_i$ already holds a shared lock on $Q$. If it does, then the system issues an upgrade($Q$) instruction, followed by the write($Q$) instruction. Otherwise, the system issues a lock-X($Q$) instruction, followed by the write($Q$) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

### 15.1.4 Implementation of Locking

A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks). Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

Figure 15.10 shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.

Although the figure does not show it, the lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiently the set of locks held by a given transaction.

The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

  It always grants a lock request on a data item that is not currently locked. But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise the request has to wait.

- When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.

- If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction (see Section 16.3), it releases all locks held by the aborted transaction.

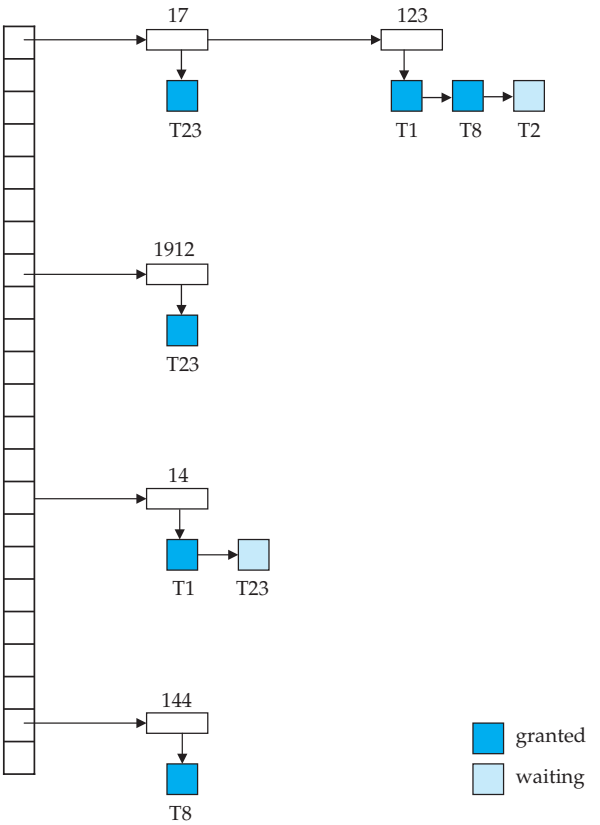



**Figure 15.10** Lock table.

This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted. We study how to detect and handle deadlocks later, in Section 15.2.2. Section 17.2.1 describes an alternative implementation—one that uses shared memory instead of message passing for lock request/grant.

### 15.1.5 Graph-Based Protocols

As noted in Section 15.1.3, if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database. There are various models that can give us the additional information, each differing in the amount of information provided. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

To acquire such prior knowledge, we impose a partial ordering $\rightarrow$ on the set $\mathbf{D} = \{d_1, d_2, \ldots, d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing

both $d_i$ and $d_j$ must access $d_i$ before accessing $d_j$. This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

The partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a **database graph**. In this section, for the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees. We shall present a simple protocol, called the *tree protocol*, which is restricted to employ only *exclusive* locks. References to other, more complex, graph-based locking protocols are in the bibliographical notes.

In the **tree protocol**, the only lock instruction allowed is lock-x. Each transaction $T_i$ can lock a data item at most once, and must observe the following rules:

1. The first lock by $T_i$ may be on any data item.
2. Subsequently, a data item $Q$ can be locked by $T_i$ only if the parent of $Q$ is currently locked by $T_i$.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by $T_i$ cannot subsequently be relocked by $T_i$.

All schedules that are legal under the tree protocol are conflict serializable.

To illustrate this protocol, consider the database graph of Figure 15.11. The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

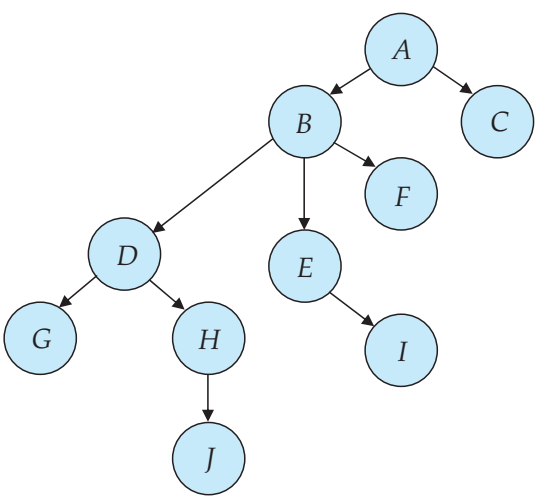



**Figure 15.11** Tree-structured database graph.

$T_{10}$: lock-X($B$); lock-X($E$); lock-X($D$); unlock($B$); unlock($E$); lock-X($G$); unlock($D$); unlock($G$).
$T_{11}$: lock-X($D$); lock-X($H$); unlock($D$); unlock($H$).
$T_{12}$: lock-X($B$); lock-X($E$); unlock($E$); unlock($B$).
$T_{13}$: lock-X($D$); lock-X($H$); unlock($D$); unlock($H$).

One possible schedule in which these four transactions participated appears in Figure 15.12. Note that, during its execution, transaction $T_{10}$ holds locks on two *disjoint* subtrees.

Observe that the schedule of Figure 15.12 is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.

The tree protocol in Figure 15.12 does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction. Holding exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency, but ensures only recoverability: For each data item with an uncommitted write, we record which transaction performed the last write to the data item. Whenever a transaction $T_i$ performs a read of an uncommitted data item, we record a **commit dependency** of $T_i$ on the

| $T_{10}$ | $T_{11}$ | $T_{12}$ | $T_{13}$ |
|---|---|---|---|
| lock-X($B$) | | | |
| | lock-X($D$) | | |
| | lock-X($H$) | | |
| | unlock($D$) | | |
| lock-X($E$) | | | |
| lock-X($D$) | | | |
| unlock($B$) | | | |
| unlock($E$) | | | |
| | | lock-X($B$) | |
| | | lock-X($E$) | |
| | unlock($H$) | | |
| lock-X($G$) | | | |
| unlock($D$) | | | |
| | | | lock-X($D$) |
| | | | lock-X($H$) |
| | | | unlock($D$) |
| | | | unlock($H$) |
| | | unlock($E$) | |
| | | unlock($B$) | |
| unlock($G$) | | | |

**Figure 15.12** Serializable schedule under the tree protocol.

transaction that performed the last write to the data item. Transaction $T_i$ is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, $T_i$ must also be aborted.

The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

However, the protocol has the disadvantage that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items $A$ and $J$ in the database graph of Figure 15.11 must lock not only $A$ and $J$, but also data items $B$, $D$, and $H$. This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocol that are not possible under the tree protocol, and vice versa. Examples of such schedules are explored in the exercises.

## 15.2 Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that $T_0$ is waiting for a data item that $T_1$ holds, and $T_1$ is waiting for a data item that $T_2$ holds, and $\ldots$, and $T_{n-1}$ is waiting for a data item that $T_n$ holds, and $T_n$ is waiting for a data item that $T_0$ holds. None of the transactions can make progress in such a situation.

The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme. As we shall see, both methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

### 15.2.1 Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. The other approach is closer to deadlock recovery, and performs transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages to this protocol: (1) it is often hard to predict, before the transaction begins, what data items need to be locked; (2) data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Another approach for preventing deadlocks is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering. We have seen one such scheme in the tree protocol, which uses a partial ordering of data items.

A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering. This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution. There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed is to ensure that locks are requested in the right order.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction $T_j$ requests a lock that transaction $T_i$ holds, the lock granted to $T_i$ may be **preempted** by rolling back of $T_i$, and granting of the lock to $T_j$. To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted. Two different deadlock-prevention schemes using timestamps have been proposed:

1. The **wait–die** scheme is a nonpreemptive technique. When transaction $T_i$ requests a data item currently held by $T_j$, $T_i$ is allowed to wait only if it has a timestamp smaller than that of $T_j$ (that is, $T_i$ is older than $T_j$). Otherwise, $T_i$ is rolled back (dies).

   For example, suppose that transactions $T_{14}$, $T_{15}$, and $T_{16}$ have timestamps 5, 10, and 15, respectively. If $T_{14}$ requests a data item held by $T_{15}$, then $T_{14}$ will wait. If $T_{24}$ requests a data item held by $T_{15}$, then $T_{16}$ will be rolled back.

2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction $T_i$ requests a data item currently held by $T_j$, $T_i$ is allowed to wait only if it has a timestamp larger than that of $T_j$ (that is, $T_i$ is younger than $T_j$). Otherwise, $T_j$ is rolled back ($T_j$ is *wounded* by $T_i$).

Returning to our example, with transactions $T_{14}$, $T_{15}$, and $T_{16}$, if $T_{14}$ requests a data item held by $T_{15}$, then the data item will be preempted from $T_{15}$, and $T_{15}$ will be rolled back. If $T_{16}$ requests a data item held by $T_{15}$, then $T_{16}$ will wait.

The major problem with both of these schemes is that unnecessary rollbacks may occur.

Another simple approach to deadlock prevention is based on **lock timeouts**. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery, which Section 15.2.2 discusses.

The timeout scheme is particularly easy to implement, and works well if transactions are short and if long waits are likely to be due to deadlocks. However, in general it is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources. Starvation is also a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

### 15.2.2 Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

In this section, we elaborate on these issues.

#### 15.2.2.1 Deadlock Detection

Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. This graph consists of a pair $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set $E$ of edges is an ordered pair $T_i \rightarrow T_j$. If $T_i \rightarrow T_j$ is in $E$,

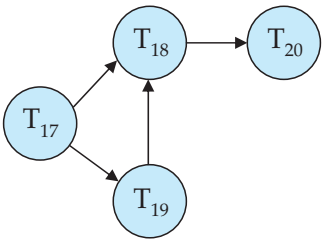



**Figure 15.13** Wait-for graph with no cycle.

then there is a directed edge from transaction $T_i$ to $T_j$, implying that transaction $T_i$ is waiting for transaction $T_j$ to release a data item that it needs.

When transaction $T_i$ requests a data item currently being held by transaction $T_j$, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction $T_j$ is no longer holding a data item needed by transaction $T_i$.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the wait-for graph in Figure 15.13, which depicts the following situation:

- Transaction $T_{17}$ is waiting for transactions $T_{18}$ and $T_{19}$.
- Transaction $T_{19}$ is waiting for transaction $T_{18}$.
- Transaction $T_{18}$ is waiting for transaction $T_{20}$.

Since the graph has no cycle, the system is not in a deadlock state.

Suppose now that transaction $T_{20}$ is requesting an item held by $T_{19}$. The edge $T_{20} \rightarrow T_{19}$ is added to the wait-for graph, resulting in the new system state in Figure 15.14. This time, the graph contains the cycle:

$$T_{18} \rightarrow T_{20} \rightarrow T_{19} \rightarrow T_{18}$$

implying that transactions $T_{18}$, $T_{19}$, and $T_{20}$ are all deadlocked.

Consequently, the question arises: When should we invoke the detection algorithm? The answer depends on two factors:

1. How often does a deadlock occur?
2. How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently. Data items allocated to deadlocked transactions will be

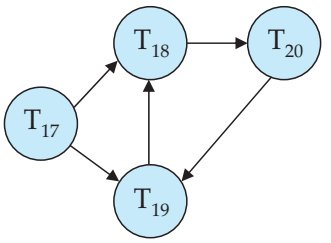



**Figure 15.14** Wait-for graph with a cycle.

unavailable to other transactions until the deadlock can be broken. In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

#### 15.2.2.2 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a victim**. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including:
   a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
   b. How many data items the transaction has used.
   c. How many more data items the transaction needs for it to complete.
   d. How many transactions will be involved in the rollback.
2. **Rollback**. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
   The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable

of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback. See the bibliographical notes for relevant references.

3. **Starvation**. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

## 15.3 Multiple Granularity

In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed.

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction $T_i$ needs to access the entire database, and a locking protocol is used, then $T_i$ must lock each item in the database. Clearly, executing these locks is time-consuming. It would be better if $T_i$ could issue a *single* lock request to lock the entire database. On the other hand, if transaction $T_j$ needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

What is needed is a mechanism to allow the system to define multiple levels of **granularity**. This is done by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Note that the tree that we describe here is significantly different from that used by the tree protocol (Section 15.1.5). A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

As an illustration, consider the tree of Figure 15.15, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type *area*; the database consists of exactly these areas. Each area in turn has nodes of type *file* as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type *record*. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction $T_i$ gets an **explicit lock** on file $F_c$ of Figure 15.15, in exclusive mode, then it has an **implicit lock** in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of $F_c$ explicitly.

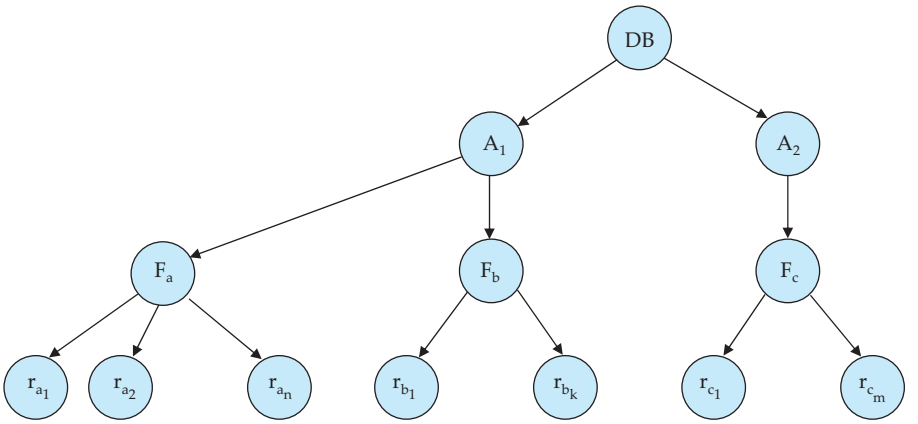



**Figure 15.15** Granularity hierarchy.

Suppose that transaction $T_j$ wishes to lock record $r_{b_6}$ of file $F_b$. Since $T_i$ has locked $F_b$ explicitly, it follows that $r_{b_6}$ is also locked (implicitly). But, when $T_j$ issues a lock request for $r_{b_6}$, $r_{b_6}$ is not explicitly locked! How does the system determine whether $T_j$ can lock $r_{b_6}$? $T_j$ must traverse the tree from the root to record $r_{b_6}$. If any node in that path is locked in an incompatible mode, then $T_j$ must be delayed.

Suppose now that transaction $T_k$ wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that $T_k$ should not succeed in locking the root node, since $T_i$ is currently holding a lock on part of the tree (specifically, on file $F_b$). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, $Q$—must traverse a path in the tree from the root to $Q$. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in Figure 15.16.

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | true | true | true | true | false |
| IX | true | true | false | false | false |
| S | true | false | true | false | false |
| SIX | true | false | false | false | false |
| X | false | false | false | false | false |

**Figure 15.16** Compatibility matrix.

The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction $T_i$ that attempts to lock a node $Q$ must follow these rules:

1. Transaction $T_i$ must observe the lock-compatibility function of Figure 15.16.
2. Transaction $T_i$ must lock the root of the tree first, and can lock it in any mode.
3. Transaction $T_i$ can lock a node $Q$ in S or IS mode only if $T_i$ currently has the parent of $Q$ locked in either IX or IS mode.
4. Transaction $T_i$ can lock a node $Q$ in X, SIX, or IX mode only if $T_i$ currently has the parent of $Q$ locked in either IX or SIX mode.
5. Transaction $T_i$ can lock a node only if $T_i$ has not previously unlocked any node (that is, $T_i$ is two phase).
6. Transaction $T_i$ can unlock a node $Q$ only if $T_i$ currently has none of the children of $Q$ locked.

Observe that the multiple-granularity protocol requires that locks be acquired in *top-down* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf-to-root) order.

As an illustration of the protocol, consider the tree of Figure 15.15 and these transactions:

- Suppose that transaction $T_{21}$ reads record $r_{a_2}$ in file $F_a$. Then, $T_{21}$ needs to lock the database, area $A_1$, and $F_a$ in IS mode (and in that order), and finally to lock $r_{a_2}$ in S mode.
- Suppose that transaction $T_{22}$ modifies record $r_{a_9}$ in file $F_a$. Then, $T_{22}$ needs to lock the database, area $A_1$, and file $F_a$ (and in that order) in IX mode, and finally to lock $r_{a_9}$ in X mode.
- Suppose that transaction $T_{23}$ reads all the records in file $F_a$. Then, $T_{23}$ needs to lock the database and area $A_1$ (and in that order) in IS mode, and finally to lock $F_a$ in S mode.

- Suppose that transaction $T_{24}$ reads the entire database. It can do so after locking the database in S mode.

We note that transactions $T_{21}$, $T_{23}$, and $T_{24}$ can access the database concurrently. Transaction $T_{22}$ can execute concurrently with $T_{21}$, but not with either $T_{23}$ or $T_{24}$.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of:

- Short transactions that access only a few data items.
- Long transactions that produce reports from an entire file or set of files.

There is a similar locking protocol that is applicable to database systems in which data granularities are organized in the form of a directed acyclic graph. See the bibliographical notes for additional references. Deadlock is possible in the multiple-granularity protocol, as it is in the two-phase locking protocol. There are techniques to reduce deadlock frequency in the multiple-granularity protocol, and also to eliminate deadlock entirely. These techniques are referenced in the bibliographical notes.

## 15.4 Timestamp-Based Protocols

The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

### 15.4.1 Timestamps

With each transaction $T_i$ in the system, we associate a unique fixed timestamp, denoted by TS($T_i$). This timestamp is assigned by the database system before the transaction $T_i$ starts execution. If a transaction $T_i$ has been assigned timestamp TS($T_i$), and a new transaction $T_j$ enters the system, then TS($T_i$) < TS($T_j$). There are two simple methods for implementing this scheme:

1. Use the value of the **system clock** as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if $TS(T_i) < TS(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction $T_i$ appears before transaction $T_j$.

To implement this scheme, we associate with each data item $Q$ two timestamp values:

- **W-timestamp**($Q$) denotes the largest timestamp of any transaction that executed write($Q$) successfully.
- **R-timestamp**($Q$) denotes the largest timestamp of any transaction that executed read($Q$) successfully.

These timestamps are updated whenever a new read($Q$) or write($Q$) instruction is executed.

### 15.4.2 The Timestamp-Ordering Protocol

The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction $T_i$ issues read($Q$).
   a. If $TS(T_i) <$ W-timestamp($Q$), then $T_i$ needs to read a value of $Q$ that was already overwritten. Hence, the read operation is rejected, and $T_i$ is rolled back.
   b. If $TS(T_i) \geq$ W-timestamp($Q$), then the read operation is executed, and R-timestamp($Q$) is set to the maximum of R-timestamp($Q$) and $TS(T_i)$.
2. Suppose that transaction $T_i$ issues write($Q$).
   a. If $TS(T_i) <$ R-timestamp($Q$), then the value of $Q$ that $T_i$ is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls $T_i$ back.
   b. If $TS(T_i) <$ W-timestamp($Q$), then $T_i$ is attempting to write an obsolete value of $Q$. Hence, the system rejects this write operation and rolls $T_i$ back.
   c. Otherwise, the system executes the write operation and sets W-timestamp($Q$) to $TS(T_i)$.

If a transaction $T_i$ is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

To illustrate this protocol, we consider transactions $T_{25}$ and $T_{26}$. Transaction $T_{25}$ displays the contents of accounts $A$ and $B$:

$T_{25}$: read($B$);
read($A$);
display($A + B$).

Transaction $T_{26}$ transfers \$50 from account $B$ to account $A$, and then displays the contents of both:

$T_{26}$: read($B$);
$B := B - 50$;
write($B$);
read($A$);
$A := A + 50$;
write($A$);
display($A + B$).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of Figure 15.17, TS($T_{25}$) < TS($T_{26}$), and the schedule is possible under the timestamp protocol.

We note that the preceding execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa (see Exercise 15.29).

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order.

The protocol ensures freedom from deadlock, since no transaction ever waits. However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. If a transaction is suffering from repeated restarts, conflicting transactions need to be temporarily blocked to enable the transaction to finish.

| $T_{25}$ | $T_{26}$ |
|---|---|
| read($B$) | |
| | read($B$) |
| | $B := B - 50$ |
| | write($B$) |
| read($A$) | |
| | read($A$) |
| display($A + B$) | |
| | $A := A + 50$ |
| | write($A$) |
| | display($A + B$) |

**Figure 15.17** Schedule 3.

The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

- Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.
- Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits (see Exercise 15.30).
- Recoverability alone can be ensured by tracking uncommitted writes, and allowing a transaction $T_i$ to commit only after the commit of any transaction that wrote a value that $T_i$ read. Commit dependencies, outlined in Section 15.1.5, can be used for this purpose.

### 15.4.3 Thomas' Write Rule

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol of Section 15.4.2. Let us consider schedule 4 of Figure 15.18, and apply the timestamp-ordering protocol. Since $T_{27}$ starts before $T_{28}$, we shall assume that TS($T_{27}$) < TS($T_{28}$). The read($Q$) operation of $T_{27}$ succeeds, as does the write($Q$) operation of $T_{28}$. When $T_{27}$ attempts its write($Q$) operation, we find that TS($T_{27}$) < W-timestamp($Q$), since W-timestamp($Q$) = TS($T_{28}$). Thus, the write($Q$) by $T_{27}$ is rejected and transaction $T_{27}$ must be rolled back.

Although the rollback of $T_{27}$ is required by the timestamp-ordering protocol, it is unnecessary. Since $T_{28}$ has already written $Q$, the value that $T_{27}$ is attempting to write is one that will never need to be read. Any transaction $T_i$ with TS($T_i$) < TS($T_{28}$) that attempts a read($Q$) will be rolled back, since TS($T_i$) < W-timestamp($Q$). Any transaction $T_j$ with TS($T_j$) > TS($T_{28}$) must read the value of $Q$ written by $T_{28}$, rather than the value that $T_{27}$ is attempting to write.

This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol of Section 15.4.2.

| $T_{27}$ | $T_{28}$ |
|---|---|
| read($Q$) | |
| | write($Q$) |
| write($Q$) | |

**Figure 15.18** Schedule 4.

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this: Suppose that transaction $T_i$ issues write($Q$).

1. If TS($T_i$) < R-timestamp($Q$), then the value of $Q$ that $T_i$ is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls $T_i$ back.
2. If TS($T_i$) < W-timestamp($Q$), then $T_i$ is attempting to write an obsolete value of $Q$. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets W-timestamp($Q$) to TS($T_i$).

The difference between these rules and those of Section 15.4.2 lies in the second rule. The timestamp-ordering protocol requires that $T_i$ be rolled back if $T_i$ issues write($Q$) and TS($T_i$) < W-timestamp($Q$). However, here, in those cases where TS($T_i$) ≥ R-timestamp($Q$), we ignore the obsolete write.

By ignoring the write, Thomas' write rule allows schedules that are not conflict serializable but are nevertheless correct. Those non-conflict-serializable schedules allowed satisfy the definition of *view serializable* schedules (see example box). Thomas' write rule makes use of view serializability by, in effect, deleting obsolete write operations from the transactions that issue them. This modification of transactions makes it possible to generate serializable schedules that would not be possible under the other protocols presented in this chapter. For example, schedule 4 of Figure 15.18 is not conflict serializable and, thus, is not possible under the two-phase locking protocol, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write($Q$) operation of $T_{27}$ would be ignored. The result is a schedule that is *view equivalent* to the serial schedule <$T_{27}$, $T_{28}$>.

## 15.5 Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state. A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for *monitoring* the system.

The **validation protocol** requires that each transaction $T_i$ executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

### VIEW SERIALIZABILITY

There is another form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

Consider two schedules $S$ and $S'$, where the same set of transactions participates in both schedules. The schedules $S$ and $S'$ are said to be **view equivalent** if three conditions are met:

1. For each data item $Q$, if transaction $T_i$ reads the initial value of $Q$ in schedule $S$, then transaction $T_i$ must, in schedule $S'$, also read the initial value of $Q$.
2. For each data item $Q$, if transaction $T_i$ executes read($Q$) in schedule $S$, and if that value was produced by a write($Q$) operation executed by transaction $T_j$, then the read($Q$) operation of transaction $T_i$ must, in schedule $S'$, also read the value of $Q$ that was produced by the same write($Q$) operation of transaction $T_j$.
3. For each data item $Q$, the transaction (if any) that performs the final write($Q$) operation in schedule $S$ must perform the final write($Q$) operation in schedule $S'$.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule $S$ is **view serializable** if it is view equivalent to a serial schedule.

As an illustration, suppose that we augment schedule 4 with transaction $T_{29}$, and obtain the following view serializable (schedule 5):

| $T_{27}$ | $T_{28}$ | $T_{29}$ |
|---|---|---|
| read ($Q$) | | |
| | write ($Q$) | |
| write ($Q$) | | |
| | | write ($Q$) |

Indeed, schedule 5 is view equivalent to the serial schedule $<T_{27}, T_{28}, T_{29}>$, since the one read($Q$) instruction reads the initial value of $Q$ in both schedules and $T_{29}$ performs the final write of $Q$ in both schedules.

Every conflict-serializable schedule is also view serializable, but there are view-serializable schedules that are not conflict serializable. Indeed, schedule 5 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

Observe that, in schedule 5, transactions $T_{28}$ and $T_{29}$ perform write($Q$) operations without having performed a read($Q$) operation. Writes of this sort are called **blind writes**. Blind writes appear in any view-serializable schedule that is not conflict serializable.

1. **Read phase**. During this phase, the system executes transaction $T_i$. It reads the values of the various data items and stores them in variables local to $T_i$. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase**. The validation test (described below) is applied to transaction $T_i$. This determines whether $T_i$ is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.
3. **Write phase**. If the validation test succeeds for transaction $T_i$, the temporary local variables that hold the results of any write operations performed by $T_i$ are copied to the database. Read-only transactions omit this phase.

Each transaction must go through the phases in the order shown. However, phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions took place. We shall, therefore, associate three different timestamps with each transaction $T_i$:

1. **Start**($T_i$), the time when $T_i$ started its execution.
2. **Validation**($T_i$), the time when $T_i$ finished its read phase and started its validation phase.
3. **Finish**($T_i$), the time when $T_i$ finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation($T_i$). Thus, the value TS($T_i$) = Validation($T_i$) and, if TS($T_j$) < TS($T_k$), then any produced schedule must be equivalent to a serial schedule in which transaction $T_j$ appears before transaction $T_k$. The reason we have chosen Validation($T_i$), rather than Start($T_i$), as the timestamp of transaction $T_i$ is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The **validation test** for transaction $T_i$ requires that, for all transactions $T_k$ with TS($T_k$) < TS($T_i$), one of the following two conditions must hold:

1. Finish($T_k$) < Start($T_i$). Since $T_k$ completes its execution before $T_i$ started, the serializability order is indeed maintained.
2. The set of data items written by $T_k$ does not intersect with the set of data items read by $T_i$, and $T_k$ completes its write phase before $T_i$ starts its validation phase (Start($T_i$) < Finish($T_k$) < Validation($T_i$)). This condition ensures that the writes of $T_k$ and $T_i$ do not overlap. Since the writes of $T_k$ do not affect the read of $T_i$, and since $T_i$ cannot affect the read of $T_k$, the serializability order is indeed maintained.

| $T_{25}$ | $T_{26}$ |
|---|---|
| read($B$) | |
| | read($B$) |
| | $B := B - 50$ |
| | read($A$) |
| | $A := A + 50$ |
| read($A$) | |
| < validate> | |
| display($A + B$) | |
| | < validate> |
| | write($B$) |
| | write($A$) |

**Figure 15.19** Schedule 6, a schedule produced by using validation.

As an illustration, consider again transactions $T_{25}$ and $T_{26}$. Suppose that TS($T_{25}$) < TS($T_{26}$). Then, the validation phase succeeds in the schedule 6 in Figure 15.19. Note that the writes to the actual variables are performed only after the validation phase of $T_{26}$. Thus, $T_{25}$ reads the old values of $B$ and $A$, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked, to enable the long transaction to finish.

This validation scheme is called the **optimistic concurrency-control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

## 15.6 Multiversion Schemes

The concurrency-control schemes discussed thus far ensure serializability by either delaying an operation or aborting the transaction that issued the operation. For example, a read operation may be delayed because the appropriate value has not been written yet; or it may be rejected (that is, the issuing transaction must be aborted) because the value that it was supposed to read has already been overwritten. These difficulties could be avoided if old copies of each data item were kept in a system.

In **multiversion concurrency-control** schemes, each write($Q$) operation creates a new **version** of $Q$. When a transaction issues a read($Q$) operation, the

concurrency-control manager selects one of the versions of $Q$ to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

### 15.6.1 Multiversion Timestamp Ordering

The timestamp-ordering protocol can be extended to a multiversion protocol. With each transaction $T_i$ in the system, we associate a unique static timestamp, denoted by TS($T_i$). The database system assigns this timestamp before the transaction starts execution, as described in Section 15.4.

With each data item $Q$, a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$ is associated. Each version $Q_k$ contains three data fields:

- **Content** is the value of version $Q_k$.
- **W-timestamp**($Q_k$) is the timestamp of the transaction that created version $Q_k$.
- **R-timestamp**($Q_k$) is the largest timestamp of any transaction that successfully read version $Q_k$.

A transaction—say, $T_i$—creates a new version $Q_k$ of data item $Q$ by issuing a write($Q$) operation. The content field of the version holds the value written by $T_i$. The system initializes the W-timestamp and R-timestamp to TS($T_i$). It updates the R-timestamp value of $Q_k$ whenever a transaction $T_j$ reads the content of $Q_k$, and R-timestamp($Q_k$) < TS($T_j$).

The **multiversion timestamp-ordering scheme** presented next ensures serializability. The scheme operates as follows: Suppose that transaction $T_i$ issues a read($Q$) or write($Q$) operation. Let $Q_k$ denote the version of $Q$ whose write timestamp is the largest write timestamp less than or equal to TS($T_i$).

1. If transaction $T_i$ issues a read($Q$), then the value returned is the content of version $Q_k$.
2. If transaction $T_i$ issues write($Q$), and if TS($T_i$) < R-timestamp($Q_k$), then the system rolls back transaction $T_i$. On the other hand, if TS($T_i$) = W-timestamp($Q_k$), the system overwrites the contents of $Q_k$; otherwise (if TS($T_i$) > R-timestamp($Q_k$)), it creates a new version of $Q$.

The justification for rule 1 is clear. A transaction reads the most recent version that comes before it in time. The second rule forces a transaction to abort if it is "too late" in doing a write. More precisely, if $T_i$ attempts to write a version that some other transaction would have read, then we cannot allow that write to succeed.

Versions that are no longer needed are removed according to the following rule: Suppose that there are two versions, $Q_k$ and $Q_j$, of a data item, and that both

versions have a W-timestamp less than the timestamp of the oldest transaction in the system. Then, the older of the two versions $Q_k$ and $Q_j$ will not be used again, and can be deleted.

The multiversion timestamp-ordering scheme has the desirable property that a read request never fails and is never made to wait. In typical database systems, where reading is a more frequent operation than is writing, this advantage may be of major practical significance.

The scheme, however, suffers from two undesirable properties. First, the reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one. Second, the conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive. Section 15.6.2 describes an algorithm to alleviate this problem.

This multiversion timestamp-ordering scheme does not ensure recoverability and cascadelessness. It can be extended in the same manner as the basic timestamp-ordering scheme, to make it recoverable and cascadeless.

### 15.6.2 Multiversion Two-Phase Locking

The **multiversion two-phase locking protocol** attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking. This protocol differentiates between **read-only transactions** and **update transactions**.

Update transactions perform rigorous two-phase locking; that is, they hold all locks up to the end of the transaction. Thus, they can be serialized according to their commit order. Each version of a data item has a single timestamp. The timestamp in this case is not a real clock-based timestamp, but rather is a counter, which we will call the ts-counter, that is incremented during commit processing.

The database system assigns read-only transactions a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads. Thus, when a read-only transaction $T_i$ issues a read($Q$), the value returned is the contents of the version whose timestamp is the largest timestamp less than or equal to TS($T_i$).

When an update transaction reads an item, it gets a shared lock on the item, and reads the latest version of that item. When an update transaction wants to write an item, it first gets an exclusive lock on the item, and then creates a new version of the data item. The write is performed on the new version, and the timestamp of the new version is initially set to a value $\infty$, a value greater than that of any possible timestamp.

When the update transaction $T_i$ completes its actions, it carries out commit processing: First, $T_i$ sets the timestamp on every version it has created to 1 more than the value of ts-counter; then, $T_i$ increments ts-counter by 1. Only one update transaction is allowed to perform commit processing at a time.

As a result, read-only transactions that start after $T_i$ increments ts-counter will see the values updated by $T_i$, whereas those that start before $T_i$ increments ts-counter will see the value before the updates by $T_i$. In either case, read-only

transactions never need to wait for locks. Multiversion two-phase locking also ensures that schedules are recoverable and cascadeless.

Versions are deleted in a manner like that of multiversion timestamp ordering. Suppose there are two versions, $Q_k$ and $Q_j$, of a data item, and that both versions have a timestamp less than or equal to the timestamp of the oldest read-only transaction in the system. Then, the older of the two versions $Q_k$ and $Q_j$ will not be used again and can be deleted.

## 15.7 Snapshot Isolation

Snapshot isolation is a particular type of concurrency-control scheme that has gained wide acceptance in commercial and open-source systems, including Oracle, PostgreSQL, and SQL Server. We introduced snapshot isolation in Section 14.9.3. Here, we take a more detailed look into how it works.

Conceptually, snapshot isolation involves giving a transaction a "snapshot" of the database at the time when it begins its execution. It then operates on that snapshot in complete isolation from concurrent transactions. The data values in the snapshot consist only of values written by committed transactions. This isolation is ideal for read-only transactions since they never wait and are never aborted by the concurrency manager. Transactions that update the database must, of course, interact with potentially conflicting concurrent update transactions before updates are actually placed in the database. Updates are kept in the transaction's private workspace until the transaction successfully commits, at which point the updates are written to the database. When a transaction $T$ is allowed to commit, the transition of $T$ to the committed state and the writing of all of the updates made by $T$ to the database must be done as an atomic action so that any snapshot created for another transaction either includes all updates by transaction $T$ or none of them.

### 15.7.1 Validation Steps for Update Transactions

Deciding whether or not to allow an update transaction to commit requires some care. Potentially, two transactions running concurrently might both update the same data item. Since these two transactions operate in isolation using their own private snapshots, neither transaction sees the update made by the other. If both transactions are allowed to write to the database, the first update written will be overwritten by the second. The result is a **lost update**. Clearly, this must be prevented. There are two variants of snapshot isolation, both of which prevent lost updates. They are called *first committer wins* and *first updater wins*. Both approaches are based on testing the transaction against concurrent transactions. A transaction is said to be **concurrent with** $T$ if it was active or partially committed at any point from the start of $T$ up to and including the time when this test is being performed.

Under **first committer wins**, when a transaction $T$ enters the partially committed state, the following actions are taken in an atomic action:

- A test is made to see if any transaction that was concurrent with $T$ has already written an update to the database for some data item that $T$ intends to write.
- If some such transaction is found, then $T$ aborts.
- If no such transaction is found, then $T$ commits and its updates are written to the database.

This approach is called "first committer wins" because if transactions conflict, the first one to be tested using the above rule succeeds in writing its updates, while the subsequent ones are forced to abort. Details of how to implement the above tests are addressed in Exercise 15.19.

Under **first updater wins** the system uses a locking mechanism that applies only to updates (reads are unaffected by this, since they do not obtain locks). When a transaction $T_i$ attempts to update a data item, it requests a *write lock* on that data item. If the lock is not held by a concurrent transaction, the following steps are taken after the lock is acquired:

- If the item has been updated by any concurrent transaction, then $T_i$ aborts.
- Otherwise $T_i$ may proceed with its execution including possibly committing.

If, however, some other concurrent transaction $T_j$ already holds a write lock on that data item, then $T_i$ cannot proceed and the following rules are followed:

- $T_i$ waits until $T_j$ aborts or commits.
  - If $T_j$ aborts, then the lock is released and $T_i$ can obtain the lock. After the lock is acquired, the check for an update by a concurrent transaction is performed as described earlier: $T_i$ aborts if a concurrent transaction had updated the data item, and proceeds with its execution otherwise.
  - If $T_j$ commits, then $T_i$ must abort.

Locks are released when the transaction commits or aborts.

This approach is called "first updater wins" because if transactions conflict, the first one to obtain the lock is the one that is permitted to commit and perform its update. Those that attempt the update later abort unless the first updater subsequently aborts for some other reason. (As an alternative to waiting to see if the first updater $T_j$ aborts, a subsequent updater $T_i$ can be aborted as soon as it finds that the write lock it wishes to obtain is held by $T_j$.)

### 15.7.2 Serializability Issues

Snapshot isolation is attractive in practice because the overhead is low and no aborts occur unless two concurrent transactions update the same data item.

There is, however, one serious problem with the snapshot isolation scheme as we have presented it, and as it is implemented in practice: *snapshot isolation does* **not** *ensure serializability*. This is true even in Oracle, which uses snapshot isolation

as the implementation for the serializable isolation level! Next, we give examples of possible nonserializable executions under snapshot isolation and show how to deal with them.

1. Suppose that we have two concurrent transactions $T_i$ and $T_j$ and two data items $A$ and $B$. Suppose that $T_i$ reads $A$ and $B$, then updates $B$, while $T_j$ reads $A$ and $B$, then updates $A$. For simplicity, we assume there are no other concurrent transactions. Since $T_i$ and $T_j$ are concurrent, neither transaction sees the update by the other in its snapshot. But, since they update different data items, both are allowed to commit regardless of whether the system uses the first-update-wins policy or the first-committer-wins policy.

   However, the precedence graph has a cycle. There is an edge in the precedence graph from $T_i$ to $T_j$ because $T_i$ reads the value of $A$ that existed before $T_j$ writes $A$. There is also an edge in the precedence graph from $T_j$ to $T_i$ because $T_j$ reads the value of $B$ that existed before $T_i$ writes $B$. Since there is a cycle in the precedence graph, the result is a nonserializable schedule.

   This situation, where each of a pair of transactions has read data that is written by the other, but there is no data written by both transactions, is referred to as **write skew**. As a concrete example of write skew, consider a banking scenario. Suppose that the bank enforces the integrity constraint that the sum of the balances in the checking and the savings account of a customer must not be negative. Suppose the checking and savings balances for a customer are \$100 and \$200, respectively. Suppose that transaction $T_{36}$ withdraws \$200 from the checking account, after verifying the integrity constraint by reading both balances. Suppose that concurrently transaction $T_{37}$ withdraws \$200 from the savings account, again after verifying the integrity constraint. Since each of the transactions checks the integrity constraint on its own snapshot, if they run concurrently each will believe that the sum of the balances after the withdrawal is \$100, and therefore its withdrawal does not violate the constraint. Since the two transactions update different data items, they do not have any update conflict, and under snapshot isolation both of them can commit.

   Unfortunately, in the final state after both $T_{36}$ and $T_{37}$ have committed, the sum of the balances is \$-100, violating the integrity constraint. Such a violation could never have occurred in any serial execution of $T_{36}$ and $T_{37}$.

   It is worth noting that integrity constraints that are enforced by the database, such as primary-key and foreign-key constraints, cannot be checked on a snapshot; otherwise it would be possible for two concurrent transactions to insert two tuples with the same primary key value, or for a transaction to insert a foreign key value that is concurrently deleted from the referenced table. Instead, the database system must check these constraints on the current state of the database, as part of validation at the time of commit.

2. For the next example, we shall consider two concurrent update transactions that do not themselves present any problem as regards serializability unless

a read-only transaction happens to show up at just the right time to cause a problem.

Suppose that we have two concurrent transactions $T_i$ and $T_j$ and two data items $A$ and $B$. Suppose that $T_i$ reads $B$ and then updates $B$, while $T_j$ reads $A$ and $B$, then updates $A$. Running these two transactions concurrently causes no problem. Since $T_i$ accesses only data item $B$, there are no conflicts on data item $A$ and therefore there is no cycle in the precedence graph. The only edge in the precedence graph is the edge from $T_j$ to $T_i$ because $T_j$ reads the value of $B$ that existed before $T_i$ writes $B$.

However, let us suppose that $T_i$ commits while $T_j$ is still active. Suppose that, after $T_i$ commits but before $T_j$ commits, a new read-only transaction $T_k$ enters the system and $T_k$ reads both $A$ and $B$. Its snapshot includes the update by $T_i$ because $T_i$ has already committed. However, since $T_j$ has not committed, its update has not yet been written to the database and is not included in the snapshot seen by $T_k$.

Consider the edges that are added to the precedence graph on account of $T_k$. There is an edge in the precedence graph from $T_i$ to $T_k$ because $T_i$ writes the value of $B$ that existed before $T_k$ reads $B$. There is an edge in the precedence graph from $T_k$ to $T_j$ because $T_k$ reads the value of $A$ that existed before $T_j$ writes $A$. That leads to a cycle in the precedence graph, showing that the resulting schedule is nonserializable.

The above anomalies may not be as troublesome as they first appear. Recall that the reason for serializability is to ensure that, despite concurrent execution of transactions, database consistency is preserved. Since consistency is the goal, we can accept the potential for nonserializable executions if we are sure that those nonserializable executions that might occur will not lead to inconsistency. The second example above is a problem only if the application that submits the read-only transaction ($T_k$) cares about seeing updates to $A$ and $B$ out of order. In that example, we did not specify the database consistency constraints that each transaction expects to hold. If we are dealing with a financial database, it might be a very serious matter for $T_k$ to read updates out of proper serial order. On the other hand, if $A$ and $B$ are enrollments in two sections of the same course, then $T_k$ may not demand perfect serialization and we may know from our applications that update rates are low enough that any inaccuracy in what $T_k$ reads is not significant.

The fact that the database must check integrity constraints at the time of commit, and not on a snapshot, also helps avoid inconsistencies in some situations. Some financial applications create consecutive sequence numbers, for example to number bills, by taking the maximum current bill number and adding 1 to the value to get a new bill number. If two such transactions run concurrently, each would see the same set of bills in its snapshot, and each would create a new bill with the same number. Both transactions pass the validation tests for snapshot isolation, since they do not update any tuple in common. However, the execution is not serializable; the resultant database state cannot be obtained by any serial

execution of the two transactions. Creating two bills with the same number could have serious legal implications.

The above problem is an example of the phantom phenomenon, since the insert performed by each transaction conflicts with the read performed by the other transaction to find the maximum bill number, but the conflict is not detected by snapshot isolation.[1]

Luckily, in most such applications the bill number would have been declared as a primary key, and the database system would detect the primary key violation outside the snapshot, and roll back one of the two transactions.[2]

An application developer can guard against certain snapshot anomalies by appending a **for update** clause to the SQL select query as illustrated below:

```
select *
from instructor
where ID = 22222
for update;
```

Adding the **for update** clause causes the system to treat data that are read as if they had been updated for purposes of concurrency control. In our first example of write skew, if the **for update** clause is appended to the select queries that read the account balances, only one of the two concurrent transactions would be allowed to commit since it appears that both transactions have updated both the checking and savings balances.

In our second example of nonserializable execution, if the author of transaction $T_k$ wished to avoid this anomaly, the **for update** clause could be appended to the **select** query, even though there is in fact no update. In our example, if $T_k$ used **select for update**, it would be treated as if it had updated $A$ and $B$ when it read them. The result would be that either $T_k$ or $T_j$ would be aborted, and retried later as a new transaction. This would lead to a serializable execution. In this example, the queries in the other two transactions do not need the **for update** clause to be added; unnecessary use of the **for update** clause can cause significant reduction in concurrency.

Formal methods exist (see the bibliographical notes) to determine whether a given mix of transactions runs the risk of nonserializable execution under snapshot isolation, and to decide on what conflicts to introduce (using the **for update** clause, for example), to ensure serializability. Of course, such methods can work only if we know in advance what transactions are being executed. In some applications, all transactions are from a predetermined set of transactions making this analysis possible. However, if the application allows unrestricted, ad-hoc transactions, then no such analysis is possible.

[1] The SQL standard uses the term phantom problem to refer to non-repeatable predicate reads, leading some to claim that snapshot isolation avoids the phantom problem; however, such a claim is not valid under our definition of phantom conflict.

[2] The problem of duplicate bill numbers actually occurred several times in a financial application in I.I.T. Bombay, where (for reasons too complex to discuss here) the bill number was not a primary key, and was detected by financial auditors.

Of the three widely used systems that support snapshot isolation, SQL Server offers the option of a *serializable* isolation level that truly ensures serializability along with a *snapshot* isolation level that provides the performance advantages of snapshot isolation (along with the potential for the anomalies discussed above). In Oracle and PostgreSQL, the *serializable* isolation level offers only snapshot isolation.

## 15.8 Insert Operations, Delete Operations, and Predicate Reads

Until now, we have restricted our attention to read and write operations. This restriction limits transactions to data items already in the database. Some transactions require not only access to existing data items, but also the ability to create new data items. Others require the ability to delete data items. To examine how such transactions affect concurrency control, we introduce these additional operations:

- **delete**($Q$) deletes data item $Q$ from the database.
- **insert**($Q$) inserts a new data item $Q$ into the database and assigns $Q$ an initial value.

An attempt by a transaction $T_i$ to perform a read($Q$) operation after $Q$ has been deleted results in a logical error in $T_i$. Likewise, an attempt by a transaction $T_i$ to perform a read($Q$) operation before $Q$ has been inserted results in a logical error in $T_i$. It is also a logical error to attempt to delete a nonexistent data item.

### 15.8.1 Deletion

To understand how the presence of **delete** instructions affects concurrency control, we must decide when a **delete** instruction conflicts with another instruction. Let $I_i$ and $I_j$ be instructions of $T_i$ and $T_j$, respectively, that appear in schedule $S$ in consecutive order. Let $I_i$ = **delete**($Q$). We consider several instructions $I_j$.

- $I_j$ = read($Q$). $I_i$ and $I_j$ conflict. If $I_i$ comes before $I_j$, $T_j$ will have a logical error. If $I_j$ comes before $I_i$, $T_j$ can execute the read operation successfully.
- $I_j$ = write($Q$). $I_i$ and $I_j$ conflict. If $I_i$ comes before $I_j$, $T_j$ will have a logical error. If $I_j$ comes before $I_i$, $T_j$ can execute the write operation successfully.
- $I_j$ = **delete**($Q$). $I_i$ and $I_j$ conflict. If $I_i$ comes before $I_j$, $T_i$ will have a logical error. If $I_j$ comes before $I_i$, $T_i$ will have a logical error.
- $I_j$ = **insert**($Q$). $I_i$ and $I_j$ conflict. Suppose that data item $Q$ did not exist prior to the execution of $I_i$ and $I_j$. Then, if $I_i$ comes before $I_j$, a logical error results for $T_i$. If $I_j$ comes before $I_i$, then no logical error results. Likewise, if $Q$ existed prior to the execution of $I_i$ and $I_j$, then a logical error results if $I_j$ comes before $I_i$, but not otherwise.

We can conclude the following:

- Under the two-phase locking protocol, an exclusive lock is required on a data item before that item can be deleted.
- Under the timestamp-ordering protocol, a test similar to that for a write must be performed. Suppose that transaction $T_i$ issues **delete**($Q$).
  - If TS($T_i$) < R-timestamp($Q$), then the value of $Q$ that $T_i$ was to delete has already been read by a transaction $T_j$ with TS($T_j$) > TS($T_i$). Hence, the **delete** operation is rejected, and $T_i$ is rolled back.
  - If TS($T_i$) < W-timestamp($Q$), then a transaction $T_j$ with TS($T_j$) > TS($T_i$) has written $Q$. Hence, this **delete** operation is rejected, and $T_i$ is rolled back.
  - Otherwise, the **delete** is executed.

### 15.8.2 Insertion

We have already seen that an **insert**($Q$) operation conflicts with a **delete**($Q$) operation. Similarly, **insert**($Q$) conflicts with a read($Q$) operation or a write($Q$) operation; no read or write can be performed on a data item before it exists.

Since an **insert**($Q$) assigns a value to data item $Q$, an **insert** is treated similarly to a write for concurrency-control purposes:

- Under the two-phase locking protocol, if $T_i$ performs an **insert**($Q$) operation, $T_i$ is given an exclusive lock on the newly created data item $Q$.
- Under the timestamp-ordering protocol, if $T_i$ performs an **insert**($Q$) operation, the values R-timestamp($Q$) and W-timestamp($Q$) are set to TS($T_i$).

### 15.8.3 Predicate Reads and The Phantom Phenomenon

Consider transaction $T_{30}$ that executes the following SQL query on the university database:

```
select count(*)
from instructor
where dept_name = 'Physics' ;
```

Transaction $T_{30}$ requires access to all tuples of the *instructor* relation pertaining to the Physics department.

Let $T_{31}$ be a transaction that executes the following SQL insertion:

```
insert into instructor
        values (11111,'Feynman', 'Physics', 94000);
```

Let $S$ be a schedule involving $T_{30}$ and $T_{31}$. We expect there to be potential for a conflict for the following reasons:

- If $T_{30}$ uses the tuple newly inserted by $T_{31}$ in computing **count**(*), then $T_{30}$ reads a value written by $T_{31}$. Thus, in a serial schedule equivalent to $S$, $T_{31}$ must come before $T_{30}$.
- If $T_{30}$ does not use the tuple newly inserted by $T_{31}$ in computing **count**(*), then in a serial schedule equivalent to $S$, $T_{30}$ must come before $T_{31}$.

The second of these two cases is curious. $T_{30}$ and $T_{31}$ do not access any tuple in common, yet they conflict with each other! In effect, $T_{30}$ and $T_{31}$ conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. As a result, the system could fail to prevent a nonserializable schedule. This problem is called the **phantom phenomenon**.

In addition to the phantom problem, we also need to deal with the situation we saw in Section 14.10, where a transaction used an index to find only tuples with *dept_name* = "Physics", and as a result did not read any tuples with other department names. If another transaction updates one of these tuples, changing its department name to Physics, a problem equivalent to the phantom problem occurs. Both problems are rooted in predicate reads, and have a common solution.

To prevent the above problems, we allow transaction $T_{30}$ to prevent other transactions from creating new tuples in the *instructor* relation with *dept_name* = "Physics", and from updating the department name of an existing *instructor* tuple to Physics.

To find all *instructor* tuples with *dept_name* = "Physics", $T_{30}$ must search either the whole *instructor* relation, or at least an index on the relation. Up to now, we have assumed implicitly that the only data items accessed by a transaction are tuples. However, $T_{30}$ is an example of a transaction that reads information about what tuples are in a relation, and $T_{31}$ is an example of a transaction that updates that information.

Clearly, it is not sufficient merely to lock the tuples that are accessed; the information used to find the tuples that are accessed by the transaction must also be locked.

Locking of information used to find tuples can be implemented by associating a data item with the relation; the data item represents the information used to find the tuples in the relation. Transactions, such as $T_{30}$, that read the information about what tuples are in a relation would then have to lock the data item corresponding to the relation in shared mode. Transactions, such as $T_{31}$, that update the information about what tuples are in a relation would have to lock the data item in exclusive mode. Thus, $T_{30}$ and $T_{31}$ would conflict on a real data item, rather than on a phantom. Similarly, transactions that use an index to retrieve tuples must lock the index itself.

Do not confuse the locking of an entire relation, as in multiple-granularity locking, with the locking of the data item corresponding to the relation. By locking the data item, a transaction only prevents other transactions from updating information about what tuples are in the relation. Locking is still required on tuples. A transaction that directly accesses a tuple can be granted a lock on the tuples even

when another transaction has an exclusive lock on the data item corresponding to the relation itself.

The major disadvantage of locking a data item corresponding to the relation, or locking an entire index, is the low degree of concurrency— two transactions that insert different tuples into a relation are prevented from executing concurrently.

A better solution is an **index-locking** technique that avoids locking the whole index. Any transaction that inserts a tuple into a relation must insert information into every index maintained on the relation. We eliminate the phantom phenomenon by imposing a locking protocol for indices. For simplicity we shall consider only $B^+$-tree indices.

As we saw in Chapter 11, every search-key value is associated with an index leaf node. A query will usually use one or more indices to access a relation. An insert must insert the new tuple in all indices on the relation. In our example, we assume that there is an index on *instructor* for *dept_name*. Then, $T_{31}$ must modify the leaf containing the key "Physics". If $T_{30}$ reads the same leaf node to locate all tuples pertaining to the Physics department, then $T_{30}$ and $T_{31}$ conflict on that leaf node.

The **index-locking protocol** takes advantage of the availability of indices on a relation, by turning instances of the phantom phenomenon into conflicts on locks on index leaf nodes. The protocol operates as follows:

- Every relation must have at least one index.
- A transaction $T_i$ can access tuples of a relation only after first finding them through one or more of the indices on the relation. For the purpose of the index-locking protocol, a relation scan is treated as a scan through all the leaves of one of the indices.
- A transaction $T_i$ that performs a lookup (whether a range lookup or a point lookup) must acquire a shared lock on all the index leaf nodes that it accesses.
- A transaction $T_i$ may not insert, delete, or update a tuple $t_i$ in a relation $r$ without updating all indices on $r$. The transaction must obtain exclusive locks on all index leaf nodes that are affected by the insertion, deletion, or update. For insertion and deletion, the leaf nodes affected are those that contain (after insertion) or contained (before deletion) the search-key value of the tuple. For updates, the leaf nodes affected are those that (before the modification) contained the old value of the search key, and nodes that (after the modification) contain the new value of the search key.
- Locks are obtained on tuples as usual.
- The rules of the two-phase locking protocol must be observed.

Note that the index-locking protocol does not address concurrency control on internal nodes of an index; techniques for concurrency control on indices, which minimize lock conflicts, are presented in Section 15.10.

Locking an index leaf node prevents any update to the node, even if the update did not actually conflict with the predicate. A variant called key-value

locking, which minimizes such false lock conflicts, is presented in Section 15.10 as part of index concurrency control.

As noted in Section 14.10, it would appear that the existence of a conflict between transactions depends on a low-level query-processing decision by the system that is unrelated to a user-level view of the meaning of the two transactions. An alternative approach to concurrency control acquires shared locks on predicates in a query, such as the predicate "*salary* > 90000" on the *instructor* relation. Inserts and deletes of the relation must then be checked to see if they satisfy the predicate; if they do, there is a lock conflict, forcing the insert or delete to wait till the predicate lock is released. For updates, both the initial value and the final value of the tuple must be checked against the predicate. Such conflicting inserts, deletes and updates affect the set of tuples selected by the predicate, and cannot be allowed to execute concurrently with the query that acquired the (shared) predicate lock. We call the above protocol **predicate locking**;[3] predicate locking is not used in practice since it is more expensive to implement than the index-locking protocol, and does not give significant additional benefits.

Variants of the predicate-locking technique can be used for eliminating the phantom phenomenon under the other concurrency-control protocols presented in this chapter. However, many database systems, such as PostgreSQL (as of version 8.1) and (to the best of our knowledge) Oracle (as of version 10g) do not implement index locking or predicate locking, and are vulnerable to nonserializability due to phantom problems even if the isolation level is set to **serializable**.

## 15.9 Weak Levels of Consistency in Practice

In Section 14.5, we discussed the isolation levels specified by the SQL standard: serializable, repeatable read, read committed, and read uncommitted. In this section, we first briefly outline some older terminology relating to consistency levels weaker than serializability and relate it to the SQL standard levels. We then discuss the issue of concurrency control for transactions that involve user interaction, an issue that we briefly discussed earlier in Section 14.8.

### 15.9.1 Degree-Two Consistency

The purpose of **degree-two consistency** is to avoid cascading aborts without necessarily ensuring serializability. The locking protocol for degree-two consistency uses the same two lock modes that we used for the two-phase locking protocol: shared (S) and exclusive (X). A transaction must hold the appropriate lock mode when it accesses a data item, but two-phase behavior is not required.

In contrast to the situation in two-phase locking, S-locks may be released at any time, and locks may be acquired at any time. Exclusive locks, however,

[3]The term *predicate locking* was used for a version of the protocol that used shared and exclusive locks on predicates, and was thus more complicated. The version we present here, with only shared locks on predicates, is also referred to as **precision locking**.

| $T_{32}$ | $T_{33}$ |
|---|---|
| lock-S($Q$) | |
| read($Q$) | |
| unlock($Q$) | |
| | lock-X($Q$) |
| | read($Q$) |
| | write($Q$) |
| | unlock($Q$) |
| lock-S($Q$) | |
| read($Q$) | |
| unlock($Q$) | |

**Figure 15.20** Nonserializable schedule with degree-two consistency.

cannot be released until the transaction either commits or aborts. Serializability is not ensured by this protocol. Indeed, a transaction may read the same data item twice and obtain different results. In Figure 15.20, $T_{32}$ reads the value of $Q$ before and after that value is written by $T_{33}$.

Clearly, reads are not repeatable, but since exclusive locks are held until transaction commit, no transaction can read an uncommitted value. Thus, degree-two consistency is one particular implementation of the read-committed isolation level.

### 15.9.2 Cursor Stability

**Cursor stability** is a form of degree-two consistency designed for programs that iterate over tuples of a relation by using cursors. Instead of locking the entire relation, cursor stability ensures that:

- The tuple that is currently being processed by the iteration is locked in shared mode.
- Any modified tuples are locked in exclusive mode until the transaction commits.

These rules ensure that degree-two consistency is obtained. Two-phase locking is not required. Serializability is not guaranteed. Cursor stability is used in practice on heavily accessed relations as a means of increasing concurrency and improving system performance. Applications that use cursor stability must be coded in a way that ensures database consistency despite the possibility of nonserializable schedules. Thus, the use of cursor stability is limited to specialized situations with simple consistency constraints.

### 15.9.3 Concurrency Control Across User Interactions

Concurrency-control protocols usually consider transactions that do not involve user interaction. Consider the airline seat selection example from Section 14.8,

which involved user interaction. Suppose we treat all the steps from when the seat availability is initially shown to the user, till the seat selection is confirmed, as a single transaction.

If two-phase locking is used, the entire set of seats on a flight would be locked in shared mode till the user has completed the seat selection, and no other transaction would be able to update the seat allocation information in this period. Clearly such locking would be a very bad idea since a user may take a long time to make a selection, or even just abandon the transaction without explicitly cancelling it. Timestamp protocols or validation could be used instead, which avoid the problem of locking, but both these protocols would abort the transaction for a user $A$ if any other user $B$ has updated the seat allocation information, even if the seat selected by $B$ does not conflict with the seat selected by user $A$. Snapshot isolation is a good option in this situation, since it would not abort the transaction of user $A$ as long as $B$ did not select the same seat as $A$.

However, snapshot isolation requires the database to remember information about updates performed by a transaction even after it has committed, as long as any other concurrent transaction is still active, which can be problematic for long duration transactions.

Another option is to split a transaction that involves user interaction into two or more transactions, such that no transaction spans a user interaction. If our seat selection transaction is split thus, the first transaction would read the seat availability, while the second transaction would complete the allocation of the selected seat. If the second transaction is written carelessly, it could assign the selected seat to the user, without checking if the seat was meanwhile assigned to some other user, resulting in a lost-update problem. To avoid the problem, as we outlined in Section 14.8, the second transaction should perform the seat allocation only if the seat was not meanwhile assigned to some other user.

The above idea has been generalized in an alternative concurrency control scheme, which uses version numbers stored in tuples to avoid lost updates. The schema of each relation is altered by adding an extra *version_number* attribute, which is initialized to 0 when the tuple is created. When a transaction reads (for the first time) a tuple that it intends to update, it remembers the version number of that tuple. The read is performed as a stand-alone transaction on the database, and hence any locks that may be obtained are released immediately. Updates are done locally, and copied to the database as part of commit processing, using the following steps which are executed atomically (that is, as part of a single database transaction):

- For each updated tuple, the transaction checks if the current version number is the same as the version number of the tuple when it was first read by the transaction.
    1. If the version numbers match, the update is performed on the tuple in the database, and its version number is incremented by 1.
    2. If the version numbers do not match, the transaction is aborted, rolling back all the updates it performed.

If the version number check succeeds for all updated tuples, the transaction commits. It is worth noting that a timestamp could be used instead of the version number, without impacting the scheme in any way.

Observe the close similarity between the above scheme and snapshot isolation. The version number check implements the first-committer-wins rule used in snapshot isolation, and can be used even if the transaction was active for a very long time. However, unlike snapshot isolation, the reads performed by transaction may not correspond to a snapshot of the database; and unlike the validation-based protocol, reads performed by the transaction are not validated.

We refer to the above scheme as **optimistic concurrency control without read validation**. Optimistic concurrency control without read validation provides a weak level of serializability, and does not ensure serializability. A variant of this scheme uses version numbers to validate reads at the time of commit, in addition to validating writes, to ensure that the tuples read by the transaction were not updated subsequent to the initial read; this scheme is equivalent to the optimistic concurrency-control scheme which we saw earlier.

The above scheme has been widely used by application developers to handle transactions that involve user interaction. An attractive feature of the scheme is that it can be implemented easily on top of a database system. The validation and update steps performed as part of commit processing are then executed as a single transaction in the database, using the concurrency-control scheme of the database to ensure atomicity for commit processing. The above scheme is also used by the Hibernate object-relational mapping system (Section 9.4.2), and other object-relational mapping systems, where it is referred to as optimistic concurrency control (even though reads are not validated by default). Transactions that involve user interaction are called **conversations** in Hibernate to differentiate them from regular transactions validation using version numbers is very useful for such transactions. Object-relational mapping systems also cache database tuples in the form of objects in memory, and execute transactions on the cached objects; updates on the objects are converted into updates on the database when the transaction commits. Data may remain in cache for a long time, and if transactions update such cached data, there is a risk of lost updates. Hibernate and other object-relational mapping systems therefore perform the version number checks transparently as part of commit processing. (Hibernate allows programmers to bypass the cache and execute transactions directly on the database, if serializability is desired.)

## 15.10 Concurrency in Index Structures**

It is possible to treat access to index structures like any other database structure, and to apply the concurrency-control techniques discussed earlier. However, since indices are accessed frequently, they would become a point of great lock contention, leading to a low degree of concurrency. Luckily, indices do not have to be treated like other database structures. It is perfectly acceptable for a transaction to perform a lookup on an index twice, and to find that the structure of the index has changed in between, as long as the index lookup returns the correct set

of tuples. Thus, it is acceptable to have nonserializable concurrent access to an index, as long as the accuracy of the index is maintained.

We outline two techniques for managing concurrent access to $B^+$-trees. The bibliographical notes reference other techniques for $B^+$-trees, as well as techniques for other index structures.

The techniques that we present for concurrency control on $B^+$-trees are based on locking, but neither two-phase locking nor the tree protocol is employed. The algorithms for lookup, insertion, and deletion are those used in Chapter 11, with only minor modifications.

The first technique is called the **crabbing protocol**:

- When searching for a key value, the crabbing protocol first locks the root node in shared mode. When traversing down the tree, it acquires a shared lock on the child node to be traversed further. After acquiring the lock on the child node, it releases the lock on the parent node. It repeats this process until it reaches a leaf node.
- When inserting or deleting a key value, the crabbing protocol takes these actions:
  - It follows the same protocol as for searching until it reaches the desired leaf node. Up to this point, it obtains (and releases) only shared locks.
  - It locks the leaf node in exclusive mode and inserts or deletes the key value.
  - If it needs to split a node or coalesce it with its siblings, or redistribute key values between siblings, the crabbing protocol locks the parent of the node in exclusive mode. After performing these actions, it releases the locks on the node and siblings.

    If the parent requires splitting, coalescing, or redistribution of key values, the protocol retains the lock on the parent, and splitting, coalescing, or redistribution propagates further in the same manner. Otherwise, it releases the lock on the parent.

The protocol gets its name from the way in which crabs advance by moving sideways, moving the legs on one side, then the legs on the other, and so on alternately. The progress of locking while the protocol both goes down the tree and goes back up (in case of splits, coalescing, or redistribution) proceeds in a similar crab-like manner.

Once a particular operation releases a lock on a node, other operations can access that node. There is a possibility of deadlocks between search operations coming down the tree, and splits, coalescing, or redistribution propagating up the tree. The system can easily handle such deadlocks by restarting the search operation from the root, after releasing the locks held by the operation.

The second technique achieves even more concurrency, avoiding even holding the lock on one node while acquiring the lock on another node, by using a modified version of $B^+$-trees called **B-link trees**; B-link trees require that every

node (including internal nodes, not just the leaves) maintain a pointer to its right sibling. This pointer is required because a lookup that occurs while a node is being split may have to search not only that node but also that node's right sibling (if one exists). We shall illustrate this technique with an example later, but we first present the modified procedures of the **B-link-tree locking protocol**.

- **Lookup**. Each node of the $B^+$-tree must be locked in shared mode before it is accessed. A lock on a nonleaf node is released before any lock on any other node in the $B^+$-tree is requested. If a split occurs concurrently with a lookup, the desired search-key value may no longer appear within the range of values represented by a node accessed during lookup. In such a case, the search-key value is in the range represented by a sibling node, which the system locates by following the pointer to the right sibling. However, the system locks leaf nodes following the two-phase locking protocol, as Section 15.8.3 describes, to avoid the phantom phenomenon.
- **Insertion and deletion**. The system follows the rules for lookup to locate the leaf node into which it will make the insertion or deletion. It upgrades the shared-mode lock on this node to exclusive mode, and performs the insertion or deletion. It locks leaf nodes affected by insertion or deletion following the two-phase locking protocol, as Section 15.8.3 describes, to avoid the phantom phenomenon.
- **Split**. If the transaction splits a node, it creates a new node according to the algorithm of Section 11.3 and makes it the right sibling of the original node. The right-sibling pointers of both the original node and the new node are set. Following this, the transaction releases the exclusive lock on the original node (provided it is an internal node; leaf nodes are locked in two-phase manner), and then requests an exclusive lock on the parent, so that it can insert a pointer to the new node. (There is no need to lock or unlock the new node.)
- **Coalescence**. If a node has too few search-key values after a deletion, the node with which it will be coalesced must be locked in exclusive mode. Once the transaction has coalesced these two nodes, it requests an exclusive lock on the parent so that the deleted node can be removed. At this point, the transaction releases the locks on the coalesced nodes. Unless the parent node must be coalesced also, its lock is released.

Observe this important fact: An insertion or deletion may lock a node, unlock it, and subsequently relock it. Furthermore, a lookup that runs concurrently with a split or coalescence operation may find that the desired search key has been moved to the right-sibling node by the split or coalescence operation.

As an illustration, consider the B-link tree in Figure 15.21. Assume that there are two concurrent operations on this B-link tree:

1. Insert "Chemistry".
2. Look up "Comp. Sci."

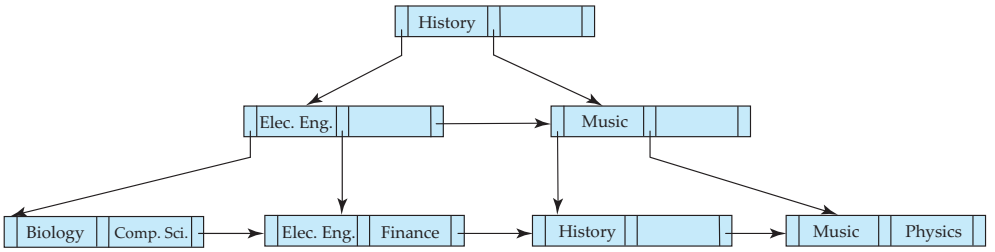



**Figure 15.21** B-link tree for *department* file with $n = 3$.

Let us assume that the insertion operation begins first. It does a lookup on "Chemistry", and finds that the node into which "Chemistry" should be inserted is full. It therefore converts its shared lock on the node to exclusive mode, and creates a new node. The original node now contains the search-key values "Biology" and "Chemistry". The new node contains the search-key value "Comp. Sci."

Now assume that a context switch occurs that results in control passing to the lookup operation. This lookup operation accesses the root, and follows the pointer to the left child of the root. It then accesses that node, and obtains a pointer to the left child. This left-child node originally contained the search-key values "Biology" and "Comp. Sci.". Since this node is currently locked by the insertion operation in exclusive mode, the lookup operation must wait. Note that, at this point, the lookup operation holds no locks at all!

The insertion operation now unlocks the leaf node and relocks its parent, this time in exclusive mode. It completes the insertion, leaving the B-link tree as in Figure 15.22. The lookup operation proceeds. However, it is holding a pointer to an incorrect leaf node. It therefore follows the right-sibling pointer to locate the next node. If this node, too, turns out to be incorrect, the lookup follows that node's right-sibling pointer. It can be shown that, if a lookup holds a pointer to an incorrect node, then, by following right-sibling pointers, the lookup must eventually reach the correct node.

Lookup and insertion operations cannot lead to deadlock. Coalescing of nodes during deletion can cause inconsistencies, since a lookup may have read a pointer to a deleted node from its parent, before the parent node was updated, and may

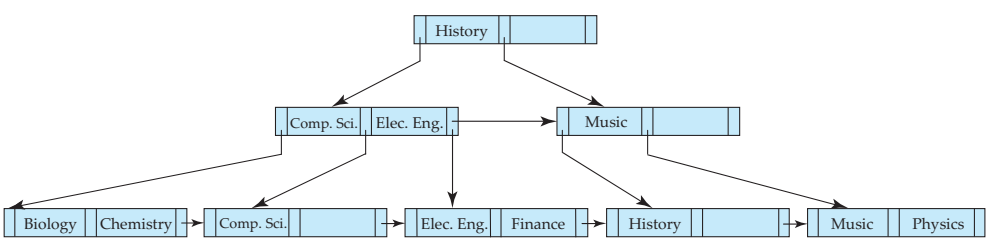



**Figure 15.22** Insertion of "Chemistry" into the B-link tree of Figure 15.21.

then try to access the deleted node. The lookup would then have to restart from the root. Leaving nodes uncoalesced avoids such inconsistencies. This solution results in nodes that contain too few search-key values and that violate some properties of $B^+$-trees. In most databases, however, insertions are more frequent than deletions, so it is likely that nodes that have too few search-key values will gain additional values relatively quickly.

Instead of locking index leaf nodes in a two-phase manner, some index concurrency-control schemes use **key-value locking** on individual key values, allowing other key values to be inserted or deleted from the same leaf. Key-value locking thus provides increased concurrency. Using key-value locking naïvely, however, would allow the phantom phenomenon to occur; to prevent the phantom phenomenon, the **next-key locking** technique is used. In this technique, every index lookup must lock not only the keys found within the range (or the single key, in case of a point lookup) but also the next-key value—that is, the key value just greater than the last key value that was within the range. Also, every insert must lock not only the value that is inserted, but also the next-key value. Thus, if a transaction attempts to insert a value that was within the range of the index lookup of another transaction, the two transactions would conflict on the key value next to the inserted key value. Similarly, deletes must also lock the next-key value to the value being deleted, to ensure that conflicts with subsequent range lookups of other queries are detected.

## 15.11 Summary

- When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.
- To ensure serializability, we can use various concurrency-control schemes. All these schemes either delay an operation or abort the transaction that issued the operation. The most common ones are locking protocols, timestamp-ordering schemes, validation techniques, and multiversion schemes.
- A locking protocol is a set of rules that state when a transaction may lock and unlock each of the data items in the database.
- The two-phase locking protocol allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item. The protocol ensures serializability, but not deadlock freedom. In the absence of information concerning the manner in which data items are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability.
- The strict two-phase locking protocol permits release of exclusive locks only at the end of transaction, in order to ensure recoverability and cascadelessness

of the resulting schedules. The rigorous two-phase locking protocol releases all locks only at the end of the transaction.

- Graph-based locking protocols impose restrictions on the order in which items are accessed, and can thereby ensure serializability without requiring the use of two-phase locking, and can additionally ensure deadlock freedom.
- Various locking protocols do not guard against deadlocks. One way to prevent deadlock is to use an ordering of data items, and to request locks in a sequence consistent with the ordering.
- Another way to prevent deadlock is to use preemption and transaction rollbacks. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps to decide whether a transaction should wait or roll back. If a transaction is rolled back, it retains its old timestamp when restarted. The wound–wait scheme is a preemptive scheme.
- If deadlocks are not prevented, the system must deal with them by using a deadlock detection and recovery scheme. To do so, the system constructs a wait-for graph. A system is in a deadlock state if and only if the wait-for graph contains a cycle. When the deadlock detection algorithm determines that a deadlock exists, the system must recover from the deadlock. It does so by rolling back one or more transactions to break the deadlock.
- There are circumstances where it would be advantageous to group several data items, and to treat them as one aggregate data item for purposes of working, resulting in multiple levels of granularity. We allow data items of various sizes, and define a hierarchy of data items, where the small items are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Locks are acquired in root-to-leaf order; they are released in leaf-to-root order. The protocol ensures serializability, but not freedom from deadlock.
- A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction $T_i$ is smaller than the timestamp of transaction $T_j$, then the scheme ensures that the produced schedule is equivalent to a serial schedule in which transaction $T_i$ appears before transaction $T_j$. It does so by rolling back a transaction whenever such an order is violated.
- A validation scheme is an appropriate concurrency-control method in cases where a majority of transactions are read-only transactions, and thus the rate of conflicts among these transactions is low. A unique fixed timestamp is associated with each transaction in the system. The serializability order is determined by the timestamp of the transaction. A transaction in this scheme is never delayed. It must, however, pass a validation test to complete. If it does not pass the validation test, the system rolls it back to its initial state.

- A multiversion concurrency-control scheme is based on the creation of a new version of a data item for each transaction that writes that item. When a read operation is issued, the system selects one of the versions to be read. The concurrency-control scheme ensures that the version to be read is selected in a manner that ensures serializability, by using timestamps. A read operation always succeeds.
  - In multiversion timestamp ordering, a write operation may result in the rollback of the transaction.
  - In multiversion two-phase locking, write operations may result in a lock wait or, possibly, in deadlock.
- Snapshot isolation is a multiversion concurrency-control protocol based on validation, which, unlike multiversion two-phase locking, does not require transactions to be declared as read-only or update. Snapshot isolation does not guarantee serializability, but is nevertheless supported by many database systems.
- A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted. A transaction that inserts a new tuple into the database is given an exclusive lock on the tuple.
- Insertions can lead to the phantom phenomenon, in which an insertion logically conflicts with a query even though the two transactions may access no tuple in common. Such conflict cannot be detected if locking is done only on tuples accessed by the transactions. Locking is required on the data used to find the tuples in the relation. The index-locking technique solves this problem by requiring locks on certain index nodes. These locks ensure that all conflicting transactions conflict on a real data item, rather than on a phantom.
- Weak levels of consistency are used in some applications where consistency of query results is not critical, and using serializability would result in queries adversely affecting transaction processing. Degree-two consistency is one such weaker level of consistency; cursor stability is a special case of degree-two consistency, and is widely used.
- Concurrency control is a challenging task for transactions that span user interactions. Applications often implement a scheme based on validation of writes using version numbers stored in tuples; this scheme provides a weak level of serializability, and can be implemented at the application level without modifications to the database.
- Special concurrency-control techniques can be developed for special data structures. Often, special techniques are applied in $B^+$-trees to allow greater concurrency. These techniques allow nonserializable access to the $B^+$-tree, but they ensure that the $B^+$-tree structure is correct, and ensure that accesses to the database itself are serializable.

## Review Terms

- Concurrency control
- Lock types
  - Shared-mode (S) lock
  - Exclusive-mode (X) lock
- Lock
  - Compatibility
  - Request
  - Wait
  - Grant
- Deadlock
- Starvation
- Locking protocol
- Legal schedule
- Two-phase locking protocol
  - Growing phase
  - Shrinking phase
  - Lock point
  - Strict two-phase locking
  - Rigorous two-phase locking
- Lock conversion
  - Upgrade
  - Downgrade
- Graph-based protocols
  - Tree protocol
  - Commit dependency
- Deadlock handling
  - Prevention
  - Detection
  - Recovery
- Deadlock prevention
  - Ordered locking
  - Preemption of locks
  - Wait–die scheme
  - Wound–wait scheme
  - Timeout-based schemes
- Deadlock detection
  - Wait-for graph
- Deadlock recovery
  - Total rollback
  - Partial rollback
- Multiple granularity
  - Explicit locks
  - Implicit locks
  - Intention locks
- Intention lock modes
  - Intention-shared (IS)
  - Intention-exclusive (IX)
  - Shared and intention-exclusive (SIX)
- Multiple-granularity locking protocol
- Timestamp
  - System clock
  - Logical counter
  - W-timestamp($Q$)
  - R-timestamp($Q$)
- Timestamp-ordering protocol
  - Thomas' write rule
- Validation-based protocols
  - Read phase

- Validation phase
- Write phase
- Validation test

- Multiversion timestamp ordering
- Multiversion two-phase locking
  - Read-only transactions
  - Update transactions
- Snapshot isolation
  - Lost update
  - First committer wins
  - First updater wins
  - Write skew
  - Select for update
- Insert and delete operations
- Phantom phenomenon
- Index-locking protocol
- Predicate locking
- Weak levels of consistency
  - Degree-two consistency
  - Cursor stability
- Optimistic concurrency control without read validation
- Conversations
- Concurrency in indices
  - Crabbing
  - B-link trees
  - B-link-tree locking protocol
  - Next-key locking

## Practice Exercises

**15.1** Show that the two-phase locking protocol ensures conflict serializability, and that transactions can be serialized according to their lock points.

**15.2** Consider the following two transactions:

$T_{34}$: read($A$);
read($B$);
**if** $A = 0$ **then** $B := B + 1$;
write($B$).

$T_{35}$: read($B$);
read($A$);
**if** $B = 0$ **then** $A := A + 1$;
write($A$).

Add lock and unlock instructions to transactions $T_{31}$ and $T_{32}$, so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

**15.3** What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?

**15.4** Consider a database organized in the form of a rooted tree. Suppose that we insert a dummy vertex between each pair of vertices. Show that, if we follow the tree protocol on the new tree, we get better concurrency than if we follow the tree protocol on the original tree.

**15.5** Show by example that there are schedules possible under the tree protocol that are not possible under the two-phase locking protocol, and vice versa.

**15.6** Consider the following extension to the tree-locking protocol, which allows both shared and exclusive locks:

- A transaction can be either a read-only transaction, in which case it can request only shared locks, or an update transaction, in which case it can request only exclusive locks.
- Each transaction must follow the rules of the tree protocol. Read-only transactions may lock any data item first, whereas update transactions must lock the root first.

Show that the protocol ensures serializability and deadlock freedom.

**15.7** Consider the following graph-based locking protocol, which allows only exclusive lock modes, and which operates on data graphs that are in the form of a rooted directed acyclic graph.

- A transaction can lock any vertex first.
- To lock any other vertex, the transaction must be holding a lock on the majority of the parents of that vertex.

Show that the protocol ensures serializability and deadlock freedom.

**15.8** Consider the following graph-based locking protocol, which allows only exclusive lock modes and which operates on data graphs that are in the form of a rooted directed acyclic graph.

- A transaction can lock any vertex first.
- To lock any other vertex, the transaction must have visited all the parents of that vertex and must be holding a lock on one of the parents of the vertex.

Show that the protocol ensures serializability and deadlock freedom.

**15.9** Locking is not done explicitly in persistent programming languages. Rather, objects (or the corresponding pages) must be locked when the objects are accessed. Most modern operating systems allow the user to set access protections (no access, read, write) on pages, and memory access that violate the access protections result in a protection violation (see the Unix `mprotect` command, for example). Describe how the access-protection mechanism can be used for page-level locking in a persistent programming language.

| | S | X | I |
|---|---|---|---|
| S | true | false | false |
| X | false | false | false |
| I | false | false | true |

**Figure 15.23** Lock-compatibility matrix.

**15.10** Consider a database system that includes an atomic **increment** operation, in addition to the read and write operations. Let $V$ be the value of data item $X$. The operation

**increment**($X$) by $C$

sets the value of $X$ to $V + C$ in an atomic step. The value of $X$ is not available to the transaction unless the latter executes a read($X$). Figure 15.23 shows a lock-compatibility matrix for three lock modes: share mode, exclusive mode, and incrementation mode.

a. Show that, if all transactions lock the data that they access in the corresponding mode, then two-phase locking ensures serializability.

b. Show that the inclusion of **increment** mode locks allows for increased concurrency. (Hint: Consider check-clearing transactions in our bank example.)

**15.11** In timestamp ordering, **W-timestamp**($Q$) denotes the largest timestamp of any transaction that executed write($Q$) successfully. Suppose that, instead, we defined it to be the timestamp of the most recent transaction to execute write($Q$) successfully. Would this change in wording make any difference? Explain your answer.

**15.12** Use of multiple-granularity locking may require more or fewer locks than an equivalent system with a single lock granularity. Provide examples of both situations, and compare the relative amount of concurrency allowed.

**15.13** Consider the validation-based concurrency-control scheme of Section 15.5. Show that by choosing Validation($T_i$), rather than Start($T_i$), as the timestamp of transaction $T_i$, we can expect better response time, provided that conflict rates among transactions are indeed low.

**15.14** For each of the following protocols, describe aspects of practical applications that would lead you to suggest using the protocol, and aspects that would suggest not using the protocol:

- Two-phase locking.
- Two-phase locking with multiple-granularity locking.

- The tree protocol.
- Timestamp ordering.
- Validation.
- Multiversion timestamp ordering.
- Multiversion two-phase locking.

**15.15** Explain why the following technique for transaction execution may provide better performance than just using strict two-phase locking: First execute the transaction without acquiring any locks and without performing any writes to the database as in the validation-based techniques, but unlike the validation techniques do not perform either validation or writes on the database. Instead, rerun the transaction using strict two-phase locking. (Hint: Consider waits for disk I/O.)

**15.16** Consider the timestamp-ordering protocol, and two transactions, one that writes two data items $p$ and $q$, and another that reads the same two data items. Give a schedule whereby the timestamp test for a write operation fails and causes the first transaction to be restarted, in turn causing a cascading abort of the other transaction. Show how this could result in starvation of both transactions. (Such a situation, where two or more processes carry out actions, but are unable to complete their task because of interaction with the other processes, is called a **livelock**.)

**15.17** Devise a timestamp-based protocol that avoids the phantom phenomenon.

**15.18** Suppose that we use the tree protocol of Section 15.1.5 to manage concurrent access to a $B^+$-tree. Since a split may occur on an insert that affects the root, it appears that an insert operation cannot release any locks until it has completed the entire operation. Under what circumstances is it possible to release a lock earlier?

**15.19** The snapshot isolation protocol uses a validation step which, before performing a write of a data item by transaction $T$, checks if a transaction concurrent with $T$ has already written the data item.

a. A straightforward implementation uses a start timestamp and a commit timestamp for each transaction, in addition to an *update set*, that is the set of data items updated by the transaction. Explain how to perform validation for the first-committer-wins scheme by using the transaction timestamps along with the update sets. You may assume that validation and other commit processing steps are executed serially, that is for one transaction at a time,

b. Explain how the validation step can be implemented as part of commit processing for the first-committer-wins scheme, using a modification of the above scheme, where instead of using update sets, each data item has a write timestamp associated with it. Again, you

may assume that validation and other commit processing steps are executed serially.

c. The first-updater-wins scheme can be implemented using timestamps as described above, except that validation is done immediately after acquiring an exclusive lock, instead of being done at commit time.
   i. Explain how to assign write timestamps to data items to implement the first-updater-wins scheme.
   ii. Show that as a result of locking, if the validation is repeated at commit time the result would not change.
   iii. Explain why there is no need to perform validation and other commit processing steps serially in this case.

## Exercises

**15.20** What benefit does strict two-phase locking provide? What disadvantages result?

**15.21** Most implementations of database systems use strict two-phase locking. Suggest three reasons for the popularity of this protocol.

**15.22** Consider a variant of the tree protocol called the *forest* protocol. The database is organized as a forest of rooted trees. Each transaction $T_i$ must follow the following rules:

- The first lock in each tree may be on any data item.
- The second, and all subsequent, locks in a tree may be requested only if the parent of the requested node is currently locked.
- Data items may be unlocked at any time.
- A data item may not be relocked by $T_i$ after it has been unlocked by $T_i$.

Show that the forest protocol does *not* ensure serializability.

**15.23** Under what conditions is it less expensive to avoid deadlock than to allow deadlocks to occur and then to detect them?

**15.24** If deadlock is avoided by deadlock-avoidance schemes, is starvation still possible? Explain your answer.

**15.25** In multiple-granularity locking, what is the difference between implicit and explicit locking?

**15.26** Although SIX mode is useful in multiple-granularity locking, an exclusive and intention-shared (XIS) mode is of no use. Why is it useless?

**15.27** The multiple-granularity protocol rules specify that a transaction $T_i$ can lock a node $Q$ in S or IS mode only if $T_i$ currently has the parent of $Q$ locked in either IX or IS mode. Given that SIX and S locks are stronger than IX or IS locks, why does the protocol not allow locking a node in S or IS mode if the parent is locked in either SIX or S mode?

**15.28** When a transaction is rolled back under timestamp ordering, it is assigned a new timestamp. Why can it not simply keep its old timestamp?

**15.29** Show that there are schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa.

**15.30** Under a modified version of the timestamp protocol, we require that a commit bit be tested to see whether a read request must wait. Explain how the commit bit can prevent cascading abort. Why is this test not necessary for write requests?

**15.31** As discussed in Exercise 15.19, snapshot isolation can be implemented using a form of timestamp validation. However, unlike the multiversion timestamp-ordering scheme, which guarantees serializability, snapshot isolation does not guarantee serializability. Explain what is the key difference between the protocols that results in this difference.

**15.32** Outline the key similarities and differences between the timestamp based implementation of the first-committer-wins version of snapshot isolation, described in Exercise 15.19, and the optimistic-concurrency-control-without-read-validation scheme, described in Section 15.9.3.

**15.33** Explain the phantom phenomenon. Why may this phenomenon lead to an incorrect concurrent execution despite the use of the two-phase locking protocol?

**15.34** Explain the reason for the use of degree-two consistency. What disadvantages does this approach have?

**15.35** Give example schedules to show that with key-value locking, if any of lookup, insert, or delete do not lock the next-key value, the phantom phenomenon could go undetected.

**15.36** Many transactions update a common item (e.g., the cash balance at a branch), and private items (e.g., individual account balances). Explain how you can increase concurrency (and throughput) by ordering the operations of the transaction.

**15.37** Consider the following locking protocol: All items are numbered, and once an item is unlocked, only higher-numbered items may be locked. Locks may be released at any time. Only X-locks are used. Show by an example that this protocol does not guarantee serializability.

## Bibliographical Notes

Gray and Reuter [1993] provides detailed textbook coverage of transaction-processing concepts, including concurrency-control concepts and implementation details. Bernstein and Newcomer [1997] provides textbook coverage of various aspects of transaction processing including concurrency control.

The two-phase locking protocol was introduced by Eswaran et al. [1976]. The tree-locking protocol is from Silberschatz and Kedem [1980]. Other non-two-phase locking protocols that operate on more general graphs are described in Yannakakis et al. [1979], Kedem and Silberschatz [1983], and Buckley and Silberschatz [1985]. Korth [1983] explores various lock modes that can be obtained from the basic shared and exclusive lock modes.

Practice Exercise 15.4 is from Buckley and Silberschatz [1984]. Practice Exercise 15.6 is from Kedem and Silberschatz [1983]. Practice Exercise 15.7 is from Kedem and Silberschatz [1979]. Practice Exercise 15.8 is from Yannakakis et al. [1979]. Practice Exercise 15.10 is from Korth [1983].

The locking protocol for multiple-granularity data items is from Gray et al. [1975]. A detailed description is presented by Gray et al. [1976]. Kedem and Silberschatz [1983] formalizes multiple-granularity locking for an arbitrary collection of lock modes (allowing for more semantics than simply read and write). This approach includes a class of lock modes called *update* modes to deal with lock conversion. Carey [1983] extends the multiple-granularity idea to timestamp-based concurrency control. An extension of the protocol to ensure deadlock freedom is presented by Korth [1982].

The timestamp-based concurrency-control scheme is from Reed [1983]. A timestamp algorithm that does not require any rollback to ensure serializability is presented by Buckley and Silberschatz [1983]. The validation concurrency-control scheme is from Kung and Robinson [1981].

Multiversion timestamp order was introduced in Reed [1983]. A multiversion tree-locking algorithm appears in Silberschatz [1982].

Degree-two consistency was introduced in Gray et al. [1975]. The levels of consistency—or isolation—offered in SQL are explained and critiqued in Berenson et al. [1995]. Many commercial database systems use version-based approaches in combination with locking. PostgreSQL, Oracle, and SQL Server all support forms of the snapshot isolation protocol mentioned in Section 15.6.2. Details can be found in Chapters 27, 28, and 30, respectively.

It should be noted that on PostgreSQL (as of version 8.1.4) and Oracle (as of version 10g), setting the isolation level to serializable results in the use of snapshot isolation, which does not guarantee serializability. Fekete et al. [2005] describes how to ensure serializable executions under snapshot isolation, by rewriting certain transactions to introduce conflicts; these conflicts ensure that the transactions cannot run concurrently under snapshot isolation; Jorwekar et al. [2007] describes an approach, that given a set of (parametrized) transactions running under snapshot isolation, can check if the transactions are vulnerability to nonserializability,

Concurrency in $B^+$-trees was studied by Bayer and Schkolnick [1977] and Johnson and Shasha [1993]. The techniques presented in Section 15.10 are based on Kung and Lehman [1980] and Lehman and Yao [1981]. The technique of key-value locking used in ARIES provides for very high concurrency on $B^+$-tree access and is described in Mohan [1990a] and Mohan and Narang [1992]. Ellis [1987] presents a concurrency-control technique for linear hashing.

*This page intentionally left blank*

CHAPTER 16

# Recovery System

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions, introduced in Chapter 14, are preserved. An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide **high availability**; that is, it must minimize the time for which the database is not usable after a failure.

## 16.1 Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. In this chapter, we shall consider only the following types of failure:

- **Transaction failure**. There are two types of errors that may cause a transaction to fail:
  - **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be reexecuted at a later time.
- **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

- **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database. We can then propose algorithms to ensure database consistency and transaction atomicity despite failures. These algorithms, known as recovery algorithms, have two parts:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

## 16.2 Storage

As we saw in Chapter 10, the various data items in the database may be stored and accessed in a number of different storage media. In Section 14.3, we saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure. We identified three categories of storage:

- **Volatile storage**
- **Nonvolatile storage**
- **Stable storage**

Stable storage or, more accurately, an approximation thereof, plays a critical role in recovery algorithms.

### 16.2.1 Stable-Storage Implementation

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

Recall (from Chapter 10) that RAID systems guarantee that the failure of a single disk (even during data transfer) will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance.

RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off site to guard against such disasters. However, since tapes cannot be carried off site continually, updates since the most recent time that tapes were carried off site could be lost in such a disaster. More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system. Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood. We study such *remote backup* systems in Section 16.9.

In the remainder of this section, we discuss how storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in:

- **Successful completion**. The transferred information arrived safely at its destination.
- **Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.
- **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

We require that, if a **data-transfer failure** occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.
3. The output is completed only after the second write completes successfully.

If the system fails while blocks are being written, it is possible that the two copies of a block are inconsistent with each other. During recovery, for each block, the system would need to examine two copies of the blocks. If both are the same and no detectable error exists, then no further actions are necessary. (Recall that errors in a disk block, such as a partial write to the block, are detected by storing a checksum with each block.) If the system detects an error in one block, then it

replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates all copies) or results in no change.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

The protocols for writing out a block to a remote site are similar to the protocols for writing blocks to a mirrored disk system, which we examined in Chapter 10, and particularly in Practice Exercise 10.3.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies reduces the probability of a failure to even lower than two copies do, it is usually reasonable to simulate stable storage with only two copies.

### 16.2.2 Data Access

As we saw in Chapter 10, the database system resides permanently on nonvolatile storage (usually disks) and only parts of the database are in memory at any time.[1] The database is partitioned into fixed-length storage units called **blocks**. Blocks are the units of data transfer to and from disk, and may contain several data items. We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as a bank or a university.

Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**.

Block movements between disk and main memory are initiated through the following two operations:

1. input($B$) transfers the physical block $B$ to main memory.
2. output($B$) transfers the buffer block $B$ to the disk, and replaces the appropriate physical block there.

Figure 16.1 illustrates this scheme.

Conceptually, each transaction $T_i$ has a private work area in which copies of data items accessed and updated by $T_i$ are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction

[1] There is a special category of database system, called *main-memory database systems*, where the entire database can be loaded into memory at once. We consider such systems in Section 26.4.

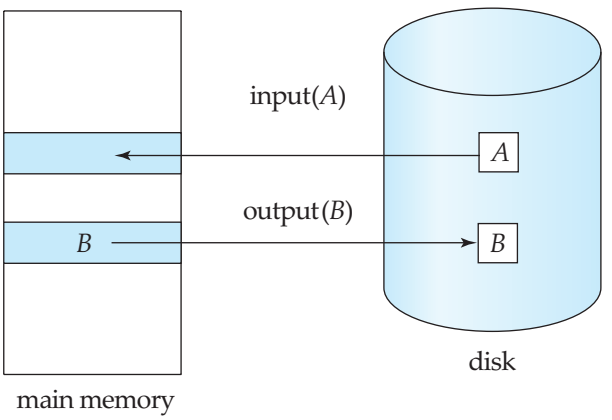



**Figure 16.1** Block storage operations.

either commits or aborts. Each data item $X$ kept in the work area of transaction $T_i$ is denoted by $x_i$. Transaction $T_i$ interacts with the database system by transferring data to and from its work area to the system buffer. We transfer data by these two operations:

1. read($X$) assigns the value of data item $X$ to the local variable $x_i$. It executes this operation as follows:
   a. If block $B_X$ on which $X$ resides is not in main memory, it issues input($B_X$).
   b. It assigns to $x_i$ the value of $X$ from the buffer block.
2. write($X$) assigns the value of local variable $x_i$ to data item $X$ in the buffer block. It executes this operation as follows:
   a. If block $B_X$ on which $X$ resides is not in main memory, it issues input($B_X$).
   b. It assigns the value of $x_i$ to $X$ in buffer $B_X$.

Note that both operations may require the transfer of a block from disk to main memory. They do not, however, specifically require the transfer of a block from main memory to disk.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to $B$ on the disk. We shall say that the database system performs a **force-output** of buffer $B$ if it issues an output($B$).

When a transaction needs to access a data item $X$ for the first time, it must execute read($X$). The system then performs all updates to $X$ on $x_i$. At any point during its execution a transaction may execute write($X$) to reflect the change to $X$ in the database itself; write($X$) must certainly be done after the final write to $X$.

The output($B_X$) operation for the buffer block $B_X$ on which $X$ resides does not need to take effect immediately after write($X$) is executed, since the block $B_X$ may contain other data items that are still being accessed. Thus, the actual output may take place later. Notice that, if the system crashes after the write($X$) operation was executed but before output($B_X$) was executed, the new value of $X$ is never written to disk and, thus, is lost. As we shall see shortly, the database system executes extra actions to ensure that updates performed by committed transactions are not lost even if there is a system crash.

## 16.3 Recovery and Atomicity

Consider again our simplified banking system and a transaction $T_i$ that transfers \$50 from account $A$ to account $B$, with initial values of $A$ and $B$ being \$1000 and \$2000, respectively. Suppose that a system crash has occurred during the execution of $T_i$, after output($B_A$) has taken place, but before output($B_B$) was executed, where $B_A$ and $B_B$ denote the buffer blocks on which $A$ and $B$ reside. Since the memory contents were lost, we do not know the fate of the transaction.

When the system restarts, the value of $A$ would be \$950, while that of $B$ would be \$2000, which is clearly inconsistent with the atomicity requirement for transaction $T_i$. Unfortunately, there is no way to find out by examining the database state what blocks had been output, and what had not, before the crash. It is possible that the transaction completed, updating the database on stable storage from an initial state with the values of $A$ and $B$ being \$1000 and \$1950; it is also possible that the transaction did not affect the stable storage at all, and the values of $A$ and $B$ were \$950 and \$2000 initially; or that the updated $B$ was output but not the updated $A$; or that the updated $A$ was output but the updated $B$ was not.

Our goal is to perform either all or no database modifications made by $T_i$. However, if $T_i$ performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output to stable storage information describing the modifications, without modifying the database itself. As we shall see, this information can help us ensure that all modifications performed by committed transactions are reflected in the database (perhaps during the course of recovery actions after a crash). This information can also help us ensure that no modifications made by an aborted transaction persist in the database.

### 16.3.1 Log Records

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database.

There are several types of log records. An **update log record** describes a single database write. It has these fields:

### SHADOW COPIES AND SHADOW PAGING

In the **shadow-copy** scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the **shadow copy**, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected. The current copy of the database is identified by a pointer, called db-pointer, which is stored on disk.

If the transaction partially commits (that is, executes its final statement) it is committed as follows: First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the fsync command for this purpose.) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted. The transaction is said to have been *committed* at the point where the updated db-pointer is written to disk.

The implementation actually depends on the write to db-pointer being atomic; that is, either all its bytes are written or none of its bytes are written. Disk systems provide atomic updates to entire blocks, or at least to a disk sector. In other words, the disk system guarantees that it will update db-pointer atomically, as long as we make sure that db-pointer lies entirely in a single sector, which we can ensure by storing db-pointer at the beginning of a block.

Shadow copy schemes are commonly used by text editors (saving the file is equivalent to transaction commit, while quitting without saving the file is equivalent to transaction abort). Shadow copying can be used for small databases, but copying a large database would be extremely expensive. A variant of shadow-copying, called **shadow-paging**, reduces copying as follows: the scheme uses a page table containing pointers to all pages; the page table itself and all updated pages are copied to a new location. Any page which is not updated by a transaction is not copied, but instead the new page table just stores a pointer to the original page. When a transaction commits, it atomically updates the pointer to the page table, which acts as db-pointer, to point to the new copy.

Shadow paging unfortunately does not work well with concurrent transactions and is not widely used in databases.

- **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier**, which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.
- **Old value**, which is the value of the data item prior to the write.

- **New value**, which is the value that the data item will have after the write.

We represent an update log record as $<T_i, X_j, V_1, V_2>$, indicating that transaction $T_i$ has performed a write on data item $X_j$. $X_j$ had value $V_1$ before the write, and has value $V_2$ after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

- $<T_i$ start$>$. Transaction $T_i$ has started.
- $<T_i$ commit$>$. Transaction $T_i$ has committed.
- $<T_i$ abort$>$. Transaction $T_i$ has aborted.

We shall introduce several other types of log records later.

Whenever a transaction performs a write, it is essential that the log record for that write be created and added to the log, before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable storage as soon as it is created. In Section 16.5, we shall see when it is safe to relax this requirement so as to reduce the overhead imposed by logging. Observe that the log contains a complete record of all database activity. As a result, the volume of data stored in the log may become unreasonably large. In Section 16.3.6, we shall show when it is safe to erase log information.

### 16.3.2 Database Modification

As we noted earlier, a transaction creates a log record prior to modifying the database. The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted; they allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk. In order for us to understand the role of these log records in recovery, we need to consider the steps a transaction takes in modifying a data item:

1. The transaction performs some computations in its own private part of main memory.
2. The transaction modifies the data block in the disk buffer in main memory holding the data item.
3. The database system executes the output operation that writes the data block to disk.

We say a transaction *modifies the database* if it performs an update on a disk buffer, or on the disk itself; updates to the private part of main memory do not count as database modifications. If a transaction does not modify the database until it has committed, it is said to use the **deferred-modification** technique. If database modifications occur while the transaction is still active, the transaction is said to use the **immediate-modification** technique. Deferred modification has the overhead that transactions need to make local copies of all updated data items; further, if a transaction reads a data item that it has updated, it must read the value from its local copy.

The recovery algorithms we describe in this chapter support immediate modification. As described, they work correctly even with deferred modification, but can be optimized to reduce overhead when used with deferred modification; we leave details as an exercise.

A recovery algorithm must take into account a variety of factors, including:

- The possibility that a transaction may have committed although some of its database modifications exist only in the disk buffer in main memory and not in the database on disk.
- The possibility that a transaction may have modified the database while in the active state and, as a result of a subsequent failure, may need to abort.

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and the new value that is to be written for the data item. This allows the system to perform *undo* and *redo* operations as appropriate.

- **Undo** using a log record sets the data item specified in the log record to the old value.
- **Redo** using a log record sets the data item specified in the log record to the new value.

### 16.3.3 Concurrency Control and Recovery

If the concurrency control scheme allows a data item $X$ that has been modified by a transaction $T_1$ to be further modified by another transaction $T_2$ before $T_1$ commits, then undoing the effects of $T_1$ by restoring the old value of $X$ (before $T_1$ updated $X$) would also undo the effects of $T_2$. To avoid such situations, recovery algorithms usually require that if a data item has been modified by a transaction, no other transaction can modify the data item until the first transaction commits or aborts.

This requirement can be ensured by acquiring an exclusive lock on any updated data item and holding the lock until the transaction commits; in other words, by using strict two-phase locking. Snapshot-isolation and validation-

based concurrency-control techniques also acquire exclusive locks on data items at the time of validation, before modifying the data items, and hold the locks until the transaction is committed; as a result the above requirement is satisfied even by these concurrency control protocols.

We discuss later, in Section 16.7, how the above requirement can be relaxed in certain cases.

When either snapshot-isolation or validation is used for concurrency control, database updates of a transaction are (conceptually) deferred until the transaction is partially committed; the deferred-modification technique is a natural fit with these concurrency control schemes. However, it is worth noting that some implementations of snapshot isolation use immediate modification, but provide a logical snapshot on demand: when a transaction needs to read an item that a concurrent transaction has updated, a copy of the (already updated) item is made, and updates made by concurrent transactions are rolled back on the copy of the item. Similarly, immediate modification of the database is a natural fit with two-phase locking, but deferred modification can also be used with two-phase locking.

### 16.3.4 Transaction Commit

We say that a transaction has **committed** when its commit log record, which is the last log record of the transaction, has been output to stable storage; at that point all earlier log records have already been output to stable storage. Thus, there is enough information in the log to ensure that even if there is a system crash, the updates of the transaction can be redone. If a system crash occurs before a log record $<$ $T_i$ commit$>$ is output to stable storage, transaction $T_i$ will be rolled back. Thus, the output of the block containing the commit log record is the single atomic action that results in a transaction getting committed.[2]

With most log-based recovery techniques, including the ones we describe in this chapter, blocks containing the data items modified by a transaction do not have to be output to stable storage when the transaction commits, but can be output some time later. We discuss this issue further in Section 16.5.2.

### 16.3.5 Using the Log to Redo and Undo Transactions

We now provide an overview of how the log can be used to recover from a system crash, and to roll back transactions during normal operation. However, we postpone details of the procedures for failure recovery and rollback to Section 16.4.

Consider our simplified banking system. Let $T_0$ be a transaction that transfers \$50 from account $A$ to account $B$:

[2]The output of a block can be made atomic by techniques for dealing with data-transfer failure, as described earlier in Section 16.2.1.

<$T_0$ start>
<$T_0$, $A$, 1000, 950>
<$T_0$, $B$, 2000, 2050>
<$T_0$ commit>
<$T_1$ start>
<$T_1$, $C$, 700, 600>
<$T_1$ commit>

**Figure 16.2** Portion of the system log corresponding to $T_0$ and $T_1$.

$T_0$: read($A$);
$A := A - 50$;
write($A$);
read($B$);
$B := B + 50$;
write($B$).

Let $T_1$ be a transaction that withdraws $100 from account $C$:

$T_1$: read($C$);
$C := C - 100$;
write($C$).

The portion of the log containing the relevant information concerning these two transactions appears in Figure 16.2.

Figure 16.3 shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of $T_0$ and $T_1$.[3]

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures. Both these procedures make use of the log to find the set of data items updated by each transaction $T_i$, and their respective old and new values.

- redo($T_i$) sets the value of all data items updated by transaction $T_i$ to the new values.

  The order in which updates are carried out by redo is important; when recovering from a system crash, if updates to a particular data item are applied in an order different from the order in which they were applied originally, the final state of that data item will have a wrong value. Most recovery algorithms, including the one we describe in Section 16.4, do not perform redo of each transaction separately; instead they perform a single scan of the log, during which redo actions are performed for each log record as it is encountered. This approach ensures the order of updates is preserved,

[3]Notice that this order could not be obtained using the deferred-modification technique, because the database is modified by $T_0$ before it commits, and likewise for $T_1$.

| Log | Database |
|---|---|
| <$T_0$ start> | |
| <$T_0$, $A$, 1000, 950> | |
| <$T_0$, $B$, 2000, 2050> | |
| | $A = 950$ |
| | $B = 2050$ |
| <$T_0$ commit> | |
| <$T_1$ start> | |
| <$T_1$, $C$, 700, 600> | |
| | $C = 600$ |
| <$T_1$ commit> | |

**Figure 16.3** State of system log and database corresponding to $T_0$ and $T_1$.

and is more efficient since the log needs to be read only once overall, instead of once per transaction.

- undo($T_i$) restores the value of all data items updated by transaction $T_i$ to the old values.

  In the recovery scheme that we describe in Section 16.4:

  - The undo operation not only restores the data items to their old value, but also writes log records to record the updates performed as part of the undo process. These log records are special **redo-only** log records, since they do not need to contain the old-value of the updated data item.

    As with the redo procedure, the order in which undo operations are performed is important; again we postpone details to Section 16.4.

  - When the undo operation for transaction $T_i$ completes, it writes a <$T_i$ abort> log record, indicating that the undo has completed.

    As we shall see in Section 16.4, the undo($T_i$) procedure is executed only once for a transaction, if the transaction is rolled back during normal processing or if on recovering from a system crash, neither a commit nor an abort record is found for transaction $T_i$. As a result, every transaction will eventually have either a commit or an abort record in the log.

After a system crash has occurred, the system consults the log to determine which transactions need to be redone, and which need to be undone so as to ensure atomicity.

- Transaction $T_i$ needs to be undone if the log contains the record <$T_i$ start>, but does not contain either the record <$T_i$ commit> or the record <$T_i$ abort>.
- Transaction $T_i$ needs to be redone if the log contains the record <$T_i$ start> and either the record <$T_i$ commit> or the record <$T_i$ abort>. It may seem strange to redo $T_i$ if the record <$T_i$ abort> is in the log. To see why this works, note

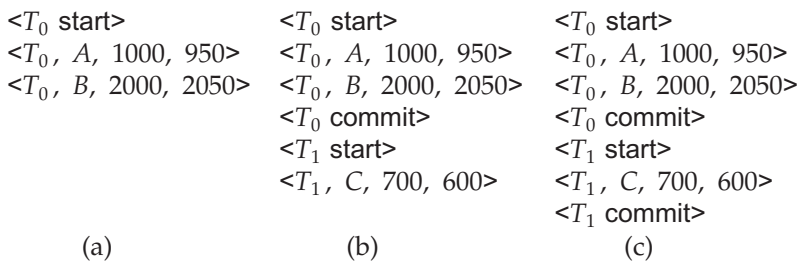



**Figure 16.4** The same log, shown at three different times.

that if $<T_i$ abort$>$ is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo $T_i$'s modifications in this case. This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time.

As an illustration, return to our banking example, with transaction $T_0$ and $T_1$ executed one after the other in the order $T_0$ followed by $T_1$. Suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases appears in Figure 16.4.

First, let us assume that the crash occurs just after the log record for the step:

write($B$)

of transaction $T_0$ has been written to stable storage (Figure 16.4a). When the system comes back up, it finds the record $<T_0$ start$>$ in the log, but no corresponding $<T_0$ commit$>$ or $<T_0$ abort$>$ record. Thus, transaction $T_0$ must be undone, so an undo($T_0$) is performed. As a result, the values in accounts $A$ and $B$ (on the disk) are restored to \$1000 and \$2000, respectively.

Next, let us assume that the crash comes just after the log record for the step:

write($C$)

of transaction $T_1$ has been written to stable storage (Figure 16.4b). When the system comes back up, two recovery actions need to be taken. The operation undo($T_1$) must be performed, since the record $<T_1$ start$>$ appears in the log, but there is no record $<T_1$ commit$>$ or $<T_1$ abort$>$. The operation redo($T_0$) must be performed, since the log contains both the record $<T_0$ start$>$ and the record $<T_0$ commit$>$. At the end of the entire recovery procedure, the values of accounts $A$, $B$, and $C$ are \$950, \$2050, and \$700, respectively.

Finally, let us assume that the crash occurs just after the log record:

$<T_1$ commit$>$

has been written to stable storage (Figure 16.4c). When the system comes back up, both $T_0$ and $T_1$ need to be redone, since the records $<T_0$ start$>$ and $<T_0$ commit$>$ appear in the log, as do the records $<T_1$ start$>$ and $<T_1$ commit$>$. After the system performs the recovery procedures redo($T_0$) and redo($T_1$), the values in accounts $A$, $B$, and $C$ are \$950, \$2050, and \$600, respectively.

### 16.3.6 Checkpoints

When a system crash occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce checkpoints.

We describe below a simple checkpoint scheme that (a) does not permit any updates to be performed while the checkpoint operation is in progress, and (b) outputs all modified buffer blocks to disk when the checkpoint is performed. We discuss later how to modify the checkpointing and recovery procedures to provide more flexibility by relaxing both these requirements.

A checkpoint is performed as follows:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record of the form $<$checkpoint $L>$, where $L$ is a list of transactions active at the time of the checkpoint.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress. We discuss how this requirement can be enforced, later, in Section 16.5.2.

The presence of a $<$checkpoint $L>$ record in the log allows the system to streamline its recovery procedure. Consider a transaction $T_i$ that completed prior to the checkpoint. For such a transaction, the $<T_i$ commit$>$ record (or $< T_i$ abort$>$ record) appears in the log before the $<$checkpoint$>$ record. Any database modifications made by $T_i$ must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on $T_i$.

After a system crash has occurred, the system examines the log to find the last $<$checkpoint $L>$ record (this can be done by searching the log backward, from the end of the log, until the first $<$checkpoint $L>$ record is found).

The redo or undo operations need to be applied only to transactions in $L$, and to all transactions that started execution after the <checkpoint $L$> record was written to the log. Let us denote this set of transactions as $T$.

- For all transactions $T_k$ in $T$ that have no <$T_k$ commit> record or <$T_k$ abort> record in the log, execute undo($T_k$).
- For all transactions $T_k$ in $T$ such that either the record <$T_k$ commit> or the record <$T_k$ abort> appears in the log, execute redo($T_k$).

Note that we need only examine the part of the log starting with the last checkpoint log record to find the set of transactions $T$, and to find out whether a commit or abort record occurs in the log for each transaction in $T$.

As an illustration, consider the set of transactions $\{T_0, T_1, \ldots, T_{100}\}$. Suppose that the most recent checkpoint took place during the execution of transaction $T_{67}$ and $T_{69}$, while $T_{68}$ and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions $T_{67}$, $T_{69}, \ldots, T_{100}$ need to be considered during the recovery scheme. Each of them needs to be redone if it has completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.

Consider the set of transactions $L$ in a checkpoint log record. For each transaction $T_i$ in $L$, log records of the transaction that occur prior to the checkpoint log record may be needed to undo the transaction, in case it does not commit. However, all log records prior to the earliest of the $< T_i$ start> log records, among transactions $T_i$ in $L$, are not needed once the checkpoint has completed. These log records can be erased whenever the database system needs to reclaim the space occupied by these records.

The requirement that transactions must not perform any updates to buffer blocks or to the log during checkpointing can be bothersome, since transaction processing has to halt while a checkpoint is in progress. A **fuzzy checkpoint** is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out. Section 16.5.4 describes fuzzy-checkpointing schemes. Later in Section 16.8 we describe a checkpoint scheme that is not only fuzzy, but does not even require all modified buffer blocks to be output to disk at the time of the checkpoint.

## 16.4 Recovery Algorithm

Until now, in discussing recovery, we have identified transactions that need to be redone and those that need to be undone, but we have not given a precise algorithm for performing these actions. We are now ready to present the full recovery algorithm using log records for recovery from transaction failure and a combination of the most recent checkpoint and log records to recover from a system crash.

The recovery algorithm described in this section requires that a data item that has been updated by an uncommitted transaction cannot be modified by any other transaction, until the first transaction has either committed or aborted. Recall that this restriction was discussed earlier, in Section 16.3.3.

### 16.4.1 Transaction Rollback

First consider transaction rollback during normal operation (that is, not during recovery from a system crash). Rollback of a transaction $T_i$ is performed as follows:

1. The log is scanned backward, and for each log record of $T_i$ of the form $<T_i, X_j, V_1, V_2>$ that is found:
   a. The value $V_1$ is written to data item $X_j$, and
   b. A special redo-only log record $<T_i, X_j, V_1>$ is written to the log, where $V_1$ is the value being restored to data item $X_j$ during the rollback. These log records are sometimes called **compensation log records**. Such records do not need undo information, since we never need to undo such an undo operation. We shall explain later how they are used.
2. Once the log record $<T_i$ start$>$ is found the backward scan is stopped, and a log record $<T_i$ abort$>$ is written to the log.

Observe that every update action performed by the transaction or on behalf of the transaction, including actions taken to restore data items to their old value, have now been recorded in the log. In Section 16.4.2 we shall see why this is a good idea.

### 16.4.2 Recovery After a System Crash

Recovery actions, when the database system is restarted after a crash, take place in two phases:

1. In the **redo phase**, the system replays updates of *all* transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred. This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back. Such incomplete transactions would either have been active at the time of the checkpoint, and thus would appear in the transaction list in the checkpoint record, or would have started later; further, such incomplete transactions would have neither a $<T_i$ abort$>$ nor a $<T_i$ commit$>$ record in the log.
   The specific steps taken while scanning the log are as follows:
   a. The list of transactions to be rolled back, undo-list, is initially set to the list $L$ in the <checkpoint $L$> log record.

b. Whenever a normal log record of the form $<T_i, X_j, V_1, V_2>$, or a redo-only log record of the form $<T_i, X_j, V_2>$ is encountered, the operation is redone; that is, the value $V_2$ is written to data item $X_j$.

c. Whenever a log record of the form $<T_i$ start$>$ is found, $T_i$ is added to undo-list.

d. Whenever a log record of the form $<T_i$ abort$>$ or $<T_i$ commit$>$ is found, $T_i$ is removed from undo-list.

At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.

2. In the **undo phase**, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.

a. Whenever it finds a log record belonging to a transaction in the undo-list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.

b. When the system finds a $<T_i$ start$>$ log record for a transaction $T_i$ in undo-list, it writes a $<T_i$ abort$>$ log record to the log, and removes $T_i$ from undo-list.

c. The undo phase terminates once undo-list becomes empty, that is, the system has found $<T_i$ start$>$ log records for all transactions that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.

Observe that the redo phase replays every log record since the most recent checkpoint record. In other words, this phase of restart recovery repeats all the update actions that were executed after the checkpoint, and whose log records reached the stable log. The actions include actions of incomplete transactions and the actions carried out to rollback failed transactions. The actions are repeated in the same order in which they were originally carried out; hence, this process is called **repeating history**. Although it may appear wasteful, repeating history even for failed transactions simplifies recovery schemes.

Figure 16.5 shows an example of actions logged during normal operation, and actions performed during failure recovery. In the log shown in the figure, transaction $T_1$ had committed, and transaction $T_0$ had been completely rolled back, before the system crashed. Observe how the value of data item $B$ is restored during the rollback of $T_0$. Observe also the checkpoint record, with the list of active transactions containing $T_0$ and $T_1$.

When recovering from a crash, in the redo phase, the system performs a redo of all operations after the last checkpoint record. In this phase, the list undo-list initially contains $T_0$ and $T_1$; $T_1$ is removed first when its commit log record is found, while $T_2$ is added when its start log record is found. Transaction $T_0$ is

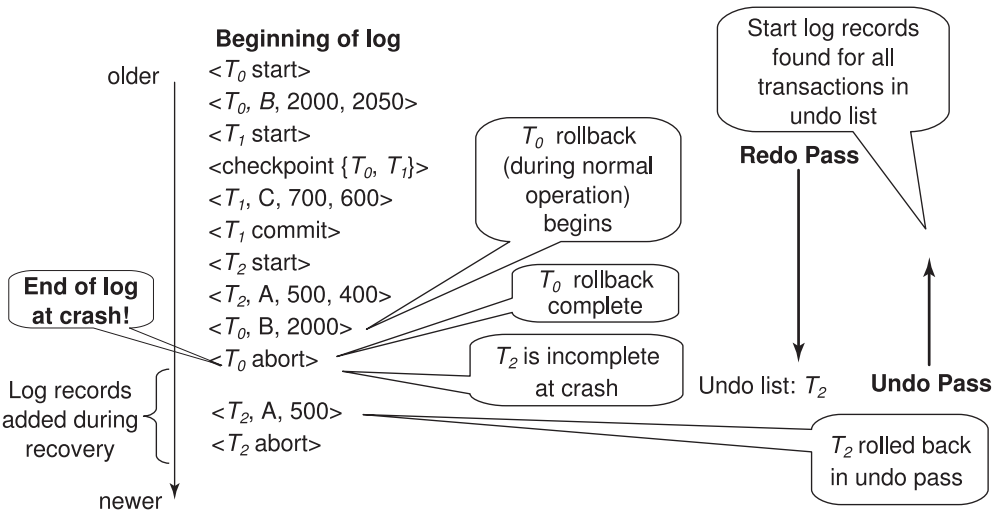



**Figure 16.5** Example of logged actions, and actions during recovery.

removed from undo-list when its abort log record is found, leaving only $T_2$ in undo-list. The undo phase scans the log backwards from the end, and when it finds a log record of $T_2$ updating $A$, the old value of $A$ is restored, and a redo-only log record written to the log. When the start record for $T_2$ is found, an abort record is added for $T_2$. Since undo-list contains no more transactions, the undo phase terminates, completing recovery.

## 16.5 Buffer Management

In this section, we consider several subtle details that are essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes a minimal amount of overhead on interactions with the database.

### 16.5.1 Log-Record Buffering

So far, we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for several reasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates to a much larger output at the physical level. Furthermore, as we saw in Section 16.2.1, the output of a block to stable storage may involve several output operations at the physical level.

The cost of outputting a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write log records to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log records can be gathered in the log buffer and output

to stable storage in a single output operation. The order of log records in the stable storage must be exactly the same as the order in which they were written to the log buffer.

As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

- Transaction $T_i$ enters the commit state after the $<T_i$ commit$>$ log record has been output to stable storage.
- Before the $<T_i$ commit$>$ log record can be output to stable storage, all log records pertaining to transaction $T_i$ must have been output to stable storage.
- Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage.

  This rule is called the **write-ahead logging (WAL)** rule. (Strictly speaking, the WAL rule requires only that the undo information in the log has been output to stable storage, and it permits the redo information to be written later. The difference is relevant in systems where undo information and redo information are stored in separate log records.)

The three rules state situations in which certain log records *must* have been output to stable storage. There is no problem resulting from the output of log records *earlier* than necessary. Thus, when the system finds it necessary to output a log record to stable storage, it outputs an entire block of log records, if there are enough log records in main memory to fill a block. If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block and are output to stable storage.

Writing the buffered log to disk is sometimes referred to as a **log force**.

### 16.5.2 Database Buffering

In Section 16.2.2, we described the use of a two-level storage hierarchy. The system stores the database in nonvolatile storage (disk), and brings blocks of data into main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block $B_1$ in main memory when another block $B_2$ needs to be brought into memory. If $B_1$ has been modified, $B_1$ must be output prior to the input of $B_2$. As discussed in Section 10.8.1 in Chapter 10, this storage hierarchy is similar to the standard operating-system concept of *virtual memory*.

One might expect that transactions would force-output all modified blocks to disk when they commit. Such a policy is called the **force** policy. The alternative, the **no-force** policy, allows a transaction to commit even if it has modified some blocks that have not yet been written back to disk. All the recovery algorithms described in this chapter work correctly even with the no-force policy. The no-

force policy allows faster commit of transactions; moreover it allows multiple updates to accumulate on a block before it is output to stable storage, which can reduce the number of output operations greatly for frequently updated blocks. As a result, the standard approach taken by most systems is the no-force policy.

Similarly, one might expect that blocks modified by a transaction that is still active should not be written to disk. This policy is called the **no-steal** policy. The alternative, the **steal** policy, allows the system to write modified blocks to disk even if the transactions that made those modifications have not all committed. As long as the write-ahead logging rule is followed, all the recovery algorithms we study in the chapter work correctly even with the steal policy. Further, the no-steal policy does not work with transactions that perform a large number of updates, since the buffer may get filled with updated pages that cannot be evicted to disk, and the transaction cannot then proceed. As a result, the standard approach taken by most systems is the steal policy.

To illustrate the need for the write-ahead logging requirement, consider our banking example with transactions $T_0$ and $T_1$. Suppose that the state of the log is:

$$<T_0 \text{ start}>$$
$$<T_0,\ A,\ 1000,\ 950>$$

and that transaction $T_0$ issues a read($B$). Assume that the block on which $B$ resides is not in main memory, and that main memory is full. Suppose that the block on which $A$ resides is chosen to be output to disk. If the system outputs this block to disk and then a crash occurs, the values in the database for accounts $A$, $B$, and $C$ are \$950, \$2000, and \$700, respectively. This database state is inconsistent. However, because of the WAL requirements, the log record:

$$<T_0,\ A,\ 1000,\ 950>$$

must be output to stable storage prior to output of the block on which $A$ resides. The system can use the log record during recovery to bring the database back to a consistent state.

When a block $B_1$ is to be output to disk, all log records pertaining to data in $B_1$ must be output to stable storage before $B_1$ is output. It is important that no writes to the block $B_1$ be in progress while the block is being output, since such a write could violate the write-ahead logging rule. We can ensure that there are no writes in progress by using a special means of locking:

- Before a transaction performs a write on a data item, it acquires an exclusive lock on the block in which the data item resides. The lock is released immediately after the update has been performed.
- The following sequence of actions is taken when a block is to be output:
  - Obtain an exclusive lock on the block, to ensure that no transaction is performing a write on the block.

- Output log records to stable storage until all log records pertaining to block $B_1$ have been output.
- Output block $B_1$ to disk.
- Release the lock once the block output has completed.

Locks on buffer blocks are unrelated to locks used for concurrency-control of transactions, and releasing them in a non-two-phase manner does not have any implications on transaction serializability. These locks, and other similar locks that are held for a short duration, are often referred to as **latches**.

Locks on buffer blocks can also be used to ensure that buffer blocks are not updated, and log records are not generated, while a checkpoint is in progress. This restriction may be enforced by acquiring exclusive locks on all buffer blocks, as well as an exclusive lock on the log, before the checkpoint operation is performed. These locks can be released as soon as the checkpoint operation has completed.

Database systems usually have a process that continually cycles through the buffer blocks, outputting modified buffer blocks back to disk. The above locking protocol must of course be followed when the blocks are output. As a result of continuous output of modified blocks, the number of **dirty blocks** in the buffer, that is, blocks that have been modified in the buffer but have not been subsequently output, is minimized. Thus, the number of blocks that have to be output during a checkpoint is minimized; further, when a block needs to be evicted from the buffer it is likely that there will be a non-dirty block available for eviction, allowing the input to proceed immediately instead of waiting for an output to complete.

### 16.5.3 Operating System Role in Buffer Management

We can manage the database buffer by using one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages. The database system manages data-block transfer in accordance with the requirements in Section 16.5.2.

   This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs. However, even when the other applications are not running, the database will not be able to make use of all the available memory. Likewise, non-database applications may not use that part of main memory reserved for the database buffer, even if some of the pages in the database buffer are not being used.

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. But, to ensure the write-ahead logging requirements in Section 16.5.1, the operating system should not write out the database buffer pages itself,

but instead should request the database system to force-output the buffer blocks. The database system in turn would force-output the buffer blocks to the database, after writing relevant log records to stable storage.

Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual-memory pages that are not currently in main memory; this space is called **swap space**. If the operating system decides to output a block $B_x$, that block is output to the swap space on disk, and there is no way for the database system to get control of the output of buffer blocks.

Therefore, if the database buffer is in virtual memory, transfers between database files and the buffer in virtual memory must be managed by the database system, which enforces the write-ahead logging requirements that we discussed.

This approach may result in extra output of data to disk. If a block $B_x$ is output by the operating system, that block is not output to the database. Instead, it is output to the swap space for the operating system's virtual memory. When the database system needs to output $B_x$, the operating system may need first to input $B_x$ from its swap space. Thus, instead of a single output of $B_x$, there may be two outputs of $B_x$ (one by the operating system, and one by the database system) and one extra input of $B_x$.

Although both approaches suffer from some drawbacks, one or the other must be chosen unless the operating system is designed to support the requirements of database logging.

### 16.5.4 Fuzzy Checkpointing

The checkpointing technique described in Section 16.3.6 requires that all updates to the database be temporarily suspended while the checkpoint is in progress. If the number of pages in the buffer is large, a checkpoint may take a long time to finish, which can result in an unacceptable interruption in processing of transactions.

To avoid such interruptions, the checkpointing technique can be modified to permit updates to start once the checkpoint record has been written, but before the modified buffer blocks are written to disk. The checkpoint thus generated is a **fuzzy checkpoint**.

Since pages are output to disk only after the checkpoint record has been written, it is possible that the system could crash before all pages are written. Thus, a checkpoint on disk may be incomplete. One way to deal with incomplete checkpoints is this: The location in the log of the checkpoint record of the last completed checkpoint is stored in a fixed position, last-checkpoint, on disk. The system does not update this information when it writes the checkpoint record. Instead, before it writes the checkpoint record, it creates a list of all modified buffer blocks. The last-checkpoint information is updated only after all buffer blocks in the list of modified buffer blocks have been output to disk.

Even with fuzzy checkpointing, a buffer block must not be updated while it is being output to disk, although other buffer blocks may be updated concurrently. The write-ahead log protocol must be followed so that (undo) log records pertaining to a block are on stable storage before the block is output.

## 16.6 Failure with Loss of Nonvolatile Storage

Until now, we have considered only the case where a failure results in the loss of information residing in volatile storage while the content of the nonvolatile storage remains intact. Although failures in which the content of nonvolatile storage is lost are rare, we nevertheless need to be prepared to deal with this type of failure. In this section, we discuss only disk storage. Our discussions apply as well to other nonvolatile storage types.

The basic scheme is to **dump** the entire contents of the database to stable storage periodically—say, once per day. For example, we may dump the database to one or more magnetic tapes. If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state.

One approach to database dumping requires that no transaction may be active during the dump procedure, and uses a procedure similar to checkpointing:

1. Output all log records currently residing in main memory onto stable storage.
2. Output all buffer blocks onto the disk.
3. Copy the contents of the database to stable storage.
4. Output a log record <dump> onto the stable storage.

Steps 1, 2, and 4 correspond to the three steps used for checkpoints in Section 16.3.6.

To recover from the loss of nonvolatile storage, the system restores the database to disk by using the most recent dump. Then, it consults the log and redoes all the actions since the most recent dump occurred. Notice that no undo operations need to be executed.

In case of a partial failure of nonvolatile storage, such as the failure of a single block or a few blocks, only those blocks need to be restored, and redo actions performed only for those blocks.

A dump of the database contents is also referred to as an **archival dump**, since we can archive the dumps and use them later to examine old states of the database. Dumps of a database and checkpointing of buffers are similar.

Most database systems also support an **SQL dump**, which writes out SQL DDL statements and SQL insert statements to a file, which can then be reexecuted to

re-create the database. Such dumps are useful when migrating data to a different instance of the database, or to a different version of the database software, since the physical locations and layout may be different in the other database instance or database software version.

The simple dump procedure described here is costly for the following two reasons. First, the entire database must be copied to stable storage, resulting in considerable data transfer. Second, since transaction processing is halted during the dump procedure, CPU cycles are wasted. **Fuzzy dump** schemes have been developed that allow transactions to be active while the dump is in progress. They are similar to fuzzy-checkpointing schemes; see the bibliographical notes for more details.

## 16.7 Early Lock Release and Logical Undo Operations

Any index used in processing a transaction, such as a $B^+$-tree, can be treated as normal data, but to increase concurrency, we can use the $B^+$-tree concurrency-control algorithm described in Section 15.10 to allow locks to be released early, in a non-two-phase manner. As a result of early lock release, it is possible that a value in a $B^+$-tree node is updated by one transaction $T_1$, which inserts an entry $(V1, R1)$, and subsequently by another transaction $T_2$, which inserts an entry $(V2, R2)$ in the same node, moving the entry $(V1, R1)$ even before $T_1$ completes execution.[4] At this point, we cannot undo transaction $T_1$ by replacing the contents of the node with the old value prior to $T_1$ performing its insert, since that would also undo the insert performed by $T_2$; transaction $T_2$ may still commit (or may have already committed). In this example, the only way to undo the effect of insertion of $(V1, R1)$ is to execute a corresponding delete operation.

In the rest of this section, we see how to extend the recovery algorithm of Section 16.4 to support early lock release.

### 16.7.1 Logical Operations

The insertion and deletion operations are examples of a class of operations that require logical undo operations since they release locks early; we call such operations **logical operations**. Such early lock release is important not only for indices, but also for operations on other system data structures that are accessed and updated very frequently; examples include data structures that track the blocks containing records of a relation, the free space in a block, and the free blocks in a database. If locks were not released early after performing operations on such data structures, transactions would tend to run serially, affecting system performance.

The theory of conflict serializability has been extended to operations, based on what operations conflict with what other operations. For example, two insert

[4]Recall that an entry consists of a key value and a record identifier, or a key value and a record in the case of the leaf level of a $B^+$-tree file organization.

operations on a B$^+$-tree do not conflict if they insert different key values, even if they both update overlapping areas of the same index page. However, insert and delete operations conflict with other insert and delete operations, as well as with read operations, if they use the same key value. See the bibliographical notes for references to more information on this topic.

Operations acquire *lower-level locks* while they execute, but release them when they complete; the corresponding transaction must however retain a *higher-level lock* in a two-phase manner to prevent concurrent transactions from executing conflicting actions. For example, while an insert operation is being performed on a B$^+$-tree page, a short-term lock is obtained on the page, allowing entries in the page to be shifted during the insert; the short-term lock is released as soon as the page has been updated. Such early lock release allows a second insert to execute on the same page. However, each transaction must obtain a lock on the key values being inserted or deleted, and retain it in a two-phase manner, to prevent a concurrent transaction from executing a conflicting read, insert or delete operation on the same key value.

Once the lower-level lock is released, the operation cannot be undone by using the old values of updated data items, and must instead be undone by executing a compensating operation; such an operation is called a **logical undo operation**. It is important that the lower-level locks acquired during an operation are sufficient to perform a subsequent logical undo of the operation, for reasons explained later in Section 16.7.4.

### 16.7.2 Logical Undo Log Records

To allow logical undo of operations, before an operation is performed to modify an index, the transaction creates a log record $<T_i, O_j,$ operation-begin$>$, where $O_j$ is a unique identifier for the operation instance.[5] While the system is executing the operation, it creates update log records in the normal fashion for all updates performed by the operation. Thus, the usual old-value and new-value information is written out as usual for each update performed by the operation; the old-value information is required in case the transaction needs to be rolled back before the operation completes. When the operation finishes, it writes an operation-end log record of the form $<T_i, O_j,$ operation-end, $U>$, where the $U$ denotes undo information.

For example, if the operation inserted an entry in a B$^+$-tree, the undo information $U$ would indicate that a deletion operation is to be performed, and would identify the B$^+$-tree and what entry to delete from the tree. Such logging of information about operations is called **logical logging**. In contrast, logging of old-value and new-value information is called **physical logging**, and the corresponding log records are called **physical log records**.

Note that in the above scheme, logical logging is used only for undo, not for redo; redo operations are performed exclusively using physical log record. This is because the state of the database after a system failure may reflect some updates

[5]The position in the log of the operation-begin log record can be used as the unique identifier.

of an operation and not other operations, depending on what buffer blocks had been written to disk before the failure. Data structures such as $B^+$-trees would not be in a consistent state, and neither logical redo nor logical undo operations can be performed on an inconsistent data structure. To perform logical redo or undo, the database state on disk must be **operation consistent**, that is, it should not have partial effects of any operation. However, as we shall see, the physical redo processing in the redo phase of the recovery scheme, along with undo processing using physical log records ensures that the parts of the database accessed by a logical undo operation are in an operation consistent state, before the logical undo operation is performed.

An operation is said to be **idempotent** if executing it several times in a row gives the same result as executing it once. Operations such as inserting an entry into a $B^+$-tree may not be idempotent, and the recovery algorithm must therefore make sure that an operation that has already been performed is not performed again. On the other hand, a physical log record is idempotent, since the corresponding data item would have the same value regardless of whether the logged update is executed one or multiple times.

### 16.7.3 Transaction Rollback With Logical Undo

When rolling back a transaction $T_i$, the log is scanned backwards, and log records corresponding to $T_i$ are processed as follows:

1. Physical log records encountered during the scan are handled as described earlier, except those that are skipped as described shortly. Incomplete logical operations are undone using the physical log records generated by the operation.
2. Completed logical operations, identified by operation-end records, are rolled back differently. Whenever the system finds a log record $<T_i, O_j,$ operation-end, $U>$, it takes special actions:
   a. It rolls back the operation by using the undo information $U$ in the log record. It logs the updates performed during the rollback of the operation just like updates performed when the operation was first executed.

      At the end of the operation rollback, instead of generating a log record $<T_i, O_j,$ operation-end, $U>$, the database system generates a log record $<T_i, O_j,$ operation-abort$>$.
   b. As the backward scan of the log continues, the system skips all log records of transaction $T_i$ until it finds the log record $<T_i, O_j,$ operation-begin$>$. After it finds the operation-begin log record, it processes log records of transaction $T_i$ in the normal manner again.

   Observe that the system logs physical undo information for the updates performed during rollback, instead of using a redo-only compensation log records. This is because a crash may occur while a logical undo is in progress,

and on recovery the system has to complete the logical undo; to do so, restart recovery will undo the partial effects of the earlier undo, using the physical undo information, and then perform the logical undo again.

Observe also that skipping over physical log records when the operation-end log record is found during rollback ensures that the old values in the physical log record are not used for rollback, once the operation completes.

3. If the system finds a record $<T_i, O_j,$ operation-abort$>$, it skips all preceding records (including the operation-end record for $O_j$) until it finds the record $<T_i, O_j,$ operation-begin$>$.

   An operation-abort log record would be found only if a transaction that is being rolled back had been partially rolled back earlier. Recall that logical operations may not be idempotent, and hence a logical undo operation must not be performed multiple times. These preceding log records must be skipped to prevent multiple rollback of the same operation, in case there had been a crash during an earlier rollback, and the transaction had already been partly rolled back.

4. As before, when the $<T_i$ start$>$ log record has been found, the transaction rollback is complete, and the system adds a record $<T_i$ abort$>$ to the log.

If a failure occurs while a logical operation is in progress, the operation-end log record for the operation will not be found when the transaction is rolled back. However, for every update performed by the operation, undo information—in the form of the old value in the physical log records—is available in the log. The physical log records will be used to roll back the incomplete operation.

Now suppose an operation undo was in progress when the system crash occurred, which could happen if a transaction was being rolled back when the crash occurred. Then the physical log records written during operation undo would be found, and the partial operation undo would itself be undone using these physical log records. Continuing in the backward scan of the log, the original operation's operation-end record would then be found, and the operation undo would be executed again. Rolling back the partial effects of the earlier undo operation using the physical log records brings the database to a consistent state, allowing the logical undo operation to be executed again.

Figure 16.6 shows an example of a log generated by two transactions, which add or subtract a value from a data item. Early lock release on the data item $C$ by transaction $T_0$ after operation $O_1$ completes allows transaction $T_1$ to update the data item using $O_2$, even before $T_0$ completes, but necessitates logical undo. The logical undo operation needs to add or subtract a value from the data item, instead of restoring an old value to the data item.

The annotations on the figure indicate that before an operation completes, rollback can perform physical undo; after the operation completes and releases lower-level locks, the undo must be performed by subtracting or adding a value, instead of restoring the old value. In the example in the figure, $T_0$ rolls back operation $O_1$ by adding 100 to $C$; on the other hand, for data item $B$, which was not subject to early lock release, undo is performed physically. Observe that $T_1$,

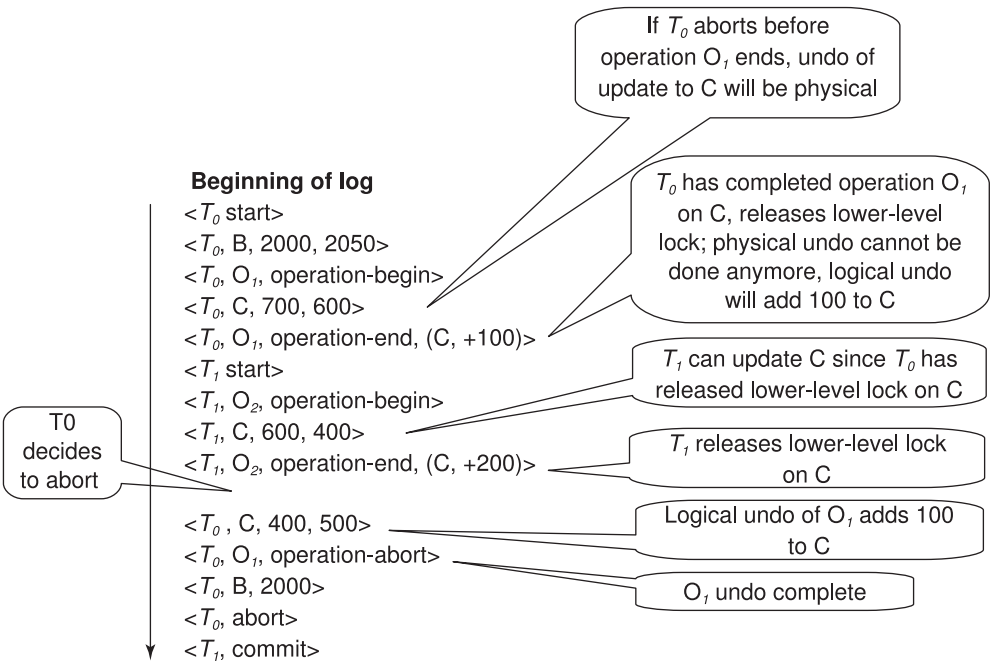



**Figure 16.6** Transaction rollback with logical undo operations.

which had performed an update on $C$ commits, and its update $O_2$, which added 200 to $C$ and was performed before the undo of $O_1$, has persisted even though $O_1$ has been undone.

Figure 16.7 shows an example of recovery from a crash, with logical undo logging. In this example, operation $T_1$ was active and executing operation $O_4$ at the time of checkpoint. In the redo pass, the actions of $O_4$ that are after the checkpoint log record are redone. At the time of crash, operation $O_5$ was being executed by $T_2$, but the operation was not complete. The undo-list contains $T_1$ and $T_2$ at the end of the redo pass. During the undo pass, the undo of operation $O_5$ is carried out using the old value in the physical log record, setting $C$ to 400; this operation is logged using a redo-only log record. The start record of $T_2$ is encountered next, resulting in the addition of $<$ $T_2$ abort$>$ to the log, and removal of $T_2$ from undo-list.

The next log record encountered is the operation-end record of $O_4$; logical undo is performed for this operation by adding 300 to $C$, which is logged physically, and an operation-abort log record is added for $O_4$. The physical log records that were part of $O_4$ are skipped until the operation-begin log record for $O_4$ is encountered. In this example, there are no other intervening log records, but in general log records from other transactions may be found before we reach the operation-begin log record; such log records should of course not be skipped (unless they are part of a completed operation for the corresponding transaction and the algorithm skips those records). After the operation-begin log record is

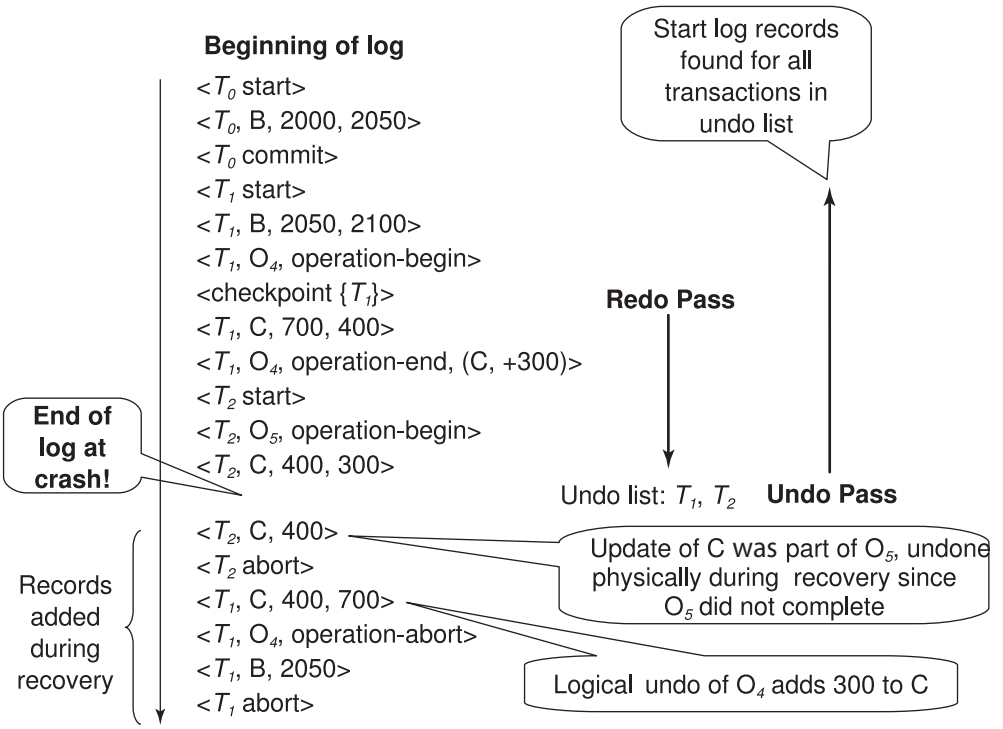



**Figure 16.7** Failure recovery actions with logical undo operations

found for $O_4$, a physical log record is found for $T_1$, which is rolled back physically. Finally the start log record for $T_1$ is found; this results in $<$ $T_1$ abort$>$ being added to the log, and $T_1$ being deleted from undo-list. At this point undo-list is empty, and the undo phase is complete.

### 16.7.4 Concurrency Issues in Logical Undo

As mentioned earlier, it is important that the lower-level locks acquired during an operation are sufficient to perform a subsequent logical undo of the operation; otherwise concurrent operations that execute during normal processing may cause problems in the undo-phase. For example, suppose the logical undo of operation $O_1$ of transaction $T_1$ can conflict at the data item level with a concurrent operation $O_2$ of transaction $T_2$, and $O_1$ completes while $O_2$ does not. Assume also that neither transaction had committed when the system crashed. The physical update log records of $O_2$ may appear before and after the operation-end record for $O_1$, and during recovery updates done during the logical undo of $O_1$ may get fully or partially overwritten by old values during the physical undo of $O_2$. This problem cannot occur if $O_1$ had obtained all the lower-level locks required for the logical undo of $O_1$, since then there cannot be such a concurrent $O_2$.

If both the original operation and its logical undo operation access a single page (such operations are called physiological operations, and are discussed in Section 16.8), the locking requirement above is met easily. Otherwise the details of the specific operation need to be considered when deciding on what lower-level locks need to be obtained. For example, update operations on a $B^+$-tree could obtain a short-term lock on the root, to ensure that operations execute serially. See the bibliographical notes for references on $B^+$-tree concurrency control and recovery exploiting logical undo logging. See the bibliographical notes also for references to an alternative approach, called multi-level recovery, which relaxes this locking requirement.

## 16.8 ARIES**

The state of the art in recovery methods is best illustrated by the ARIES recovery method. The recovery technique that we described in Section 16.4, along with the logical undo logging techniques described in Section 16.7, is modeled after ARIES, but has been simplified significantly to bring out key concepts and make it easier to understand. In contrast, ARIES uses a number of techniques to reduce the time taken for recovery, and to reduce the overhead of checkpointing. In particular, ARIES is able to avoid redoing many logged operations that have already been applied and to reduce the amount of information logged. The price paid is greater complexity; the benefits are worth the price.

The major differences between ARIES and the recovery algorithm presented earlier are that ARIES:

1. Uses a **log sequence number** (LSN) to identify log records, and stores LSNs in database pages to identify which operations have been applied to a database page.
2. Supports **physiological redo** operations, which are physical in that the affected page is physically identified, but can be logical within the page.

   For instance, the deletion of a record from a page may result in many other records in the page being shifted, if a slotted page structure (Section 10.5.2) is used. With physical redo logging, all bytes of the page affected by the shifting of records must be logged. With physiological logging, the deletion operation can be logged, resulting in a much smaller log record. Redo of the deletion operation would delete the record and shift other records as required.
3. Uses a **dirty page table** to minimize unnecessary redos during recovery. As mentioned earlier, dirty pages are those that have been updated in memory, and the disk version is not up-to-date.
4. Uses a fuzzy-checkpointing scheme that records only information about dirty pages and associated information and does not even require writing

of dirty pages to disk. It flushes dirty pages in the background, continuously, instead of writing them during checkpoints.

In the rest of this section, we provide an overview of ARIES. The bibliographical notes list references that provide a complete description of ARIES.

### 16.8.1 Data Structures

Each log record in ARIES has a log sequence number (LSN) that uniquely identifies the record. The number is conceptually just a logical identifier whose value is greater for log records that occur later in the log. In practice, the LSN is generated in such a way that it can also be used to locate the log record on disk. Typically, ARIES splits a log into multiple log files, each of which has a file number. When a log file grows to some limit, ARIES appends further log records to a new log file; the new log file has a file number that is higher by 1 than the previous log file. The LSN then consists of a file number and an offset within the file.

Each page also maintains an identifier called the **PageLSN**. Whenever an update operation (whether physical or physiological) occurs on a page, the operation stores the LSN of its log record in the PageLSN field of the page. During the redo phase of recovery, any log records with LSN less than or equal to the PageLSN of a page should not be executed on the page, since their actions are already reflected on the page. In combination with a scheme for recording PageLSNs as part of checkpointing, which we present later, ARIES can avoid even reading many pages for which logged operations are already reflected on disk. Thereby, recovery time is reduced significantly.

The PageLSN is essential for ensuring idempotence in the presence of physiological redo operations, since reapplying a physiological redo that has already been applied to a page could cause incorrect changes to a page.

Pages should not be flushed to disk while an update is in progress, since physiological operations cannot be redone on the partially updated state of the page on disk. Therefore, ARIES uses latches on buffer pages to prevent them from being written to disk while they are being updated. It releases the buffer page latch only after the update is completed, and the log record for the update has been written to the log.

Each log record also contains the LSN of the previous log record of the same transaction. This value, stored in the PrevLSN field, permits log records of a transaction to be fetched backward, without reading the whole log. There are special redo-only log records generated during transaction rollback, called **compensation log records** (**CLR**s) in ARIES. These serve the same purpose as the redo-only log records in our earlier recovery scheme. In addition CLRs serve the role of the operation-abort log records in our scheme. The CLRs have an extra field, called the UndoNextLSN, that records the LSN of the log that needs to be undone next, when the transaction is being rolled back. This field serves the same purpose as the operation identifier in the operation-abort log record in our earlier recovery scheme, which helps to skip over log records that have already been rolled back.

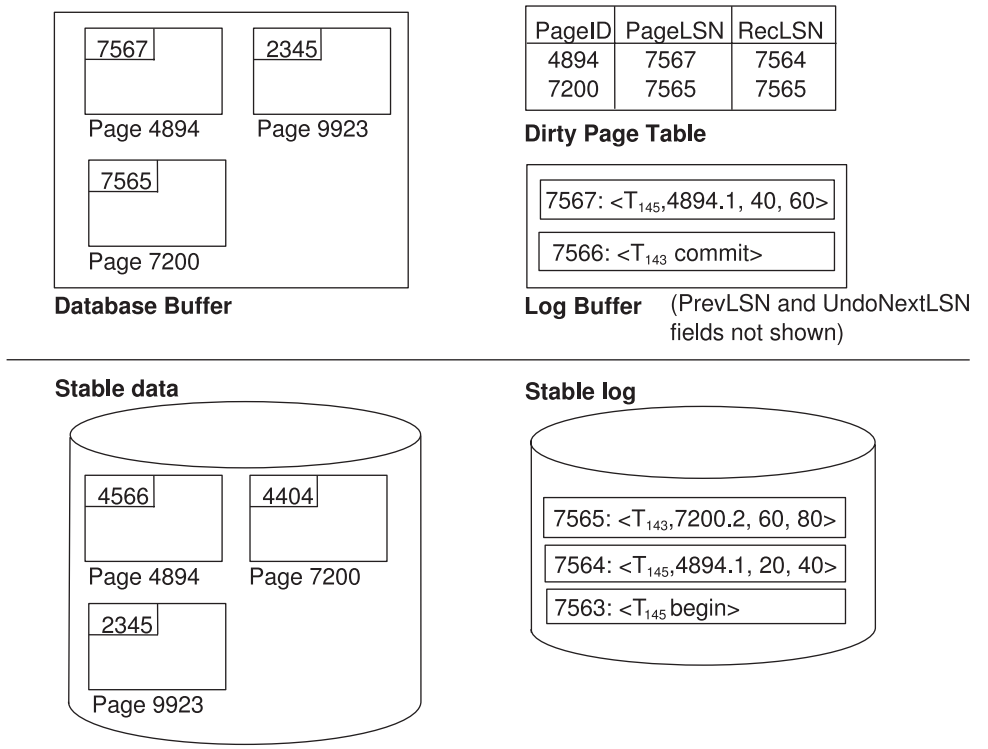



**Figure 16.8** Data structures used in ARIES.

The **DirtyPageTable** contains a list of pages that have been updated in the database buffer. For each page, it stores the PageLSN and a field called the RecLSN, which helps identify log records that have been applied already to the version of the page on disk. When a page is inserted into the DirtyPageTable (when it is first modified in the buffer pool), the value of RecLSN is set to the current end of log. Whenever the page is flushed to disk, the page is removed from the DirtyPageTable.

A **checkpoint log record** contains the DirtyPageTable and a list of active transactions. For each transaction, the checkpoint log record also notes LastLSN, the LSN of the last log record written by the transaction. A fixed position on disk also notes the LSN of the last (complete) checkpoint log record.

Figure 16.8 illustrates some of the data structures used in ARIES. The log records shown in the figure are prefixed by their LSN; these may not be explicitly stored, but inferred from the position in the log, in an actual implementation. The data item identifier in a log record is shown in two parts, for example 4894.1; the first identifies the page, and the second part identifies a record within the page (we assume a slotted page record organization within a page). Note that the log is shown with newest records on top, since older log records, which are on disk, are shown lower in the figure.

Each page (whether in the buffer or on disk) has an associated PageLSN field. You can verify that the LSN for the last log record that updated page 4894 is 7567. By comparing PageLSNs for the pages in the buffer with the PageLSNs for the corresponding pages in stable storage, you can observe that the DirtyPageTable contains entries for all pages in the buffer that have been modified since they were fetched from stable storage. The RecLSN entry in the DirtyPageTable reflects the LSN at the end of the log when the page was added to DirtyPageTable, and would be greater than or equal to the PageLSN for that page on stable storage.

### 16.8.2 Recovery Algorithm

ARIES recovers from a system crash in three passes.

- **Analysis pass**: This pass determines which transactions to undo, which pages were dirty at the time of the crash, and the LSN from which the redo pass should start.
- **Redo pass**: This pass starts from a position determined during analysis, and performs a redo, repeating history, to bring the database to a state it was in before the crash.
- **Undo pass**: This pass rolls back all transactions that were incomplete at the time of crash.

#### 16.8.2.1 Analysis Pass

The analysis pass finds the last complete checkpoint log record, and reads in the DirtyPageTable from this record. It then sets RedoLSN to the minimum of the RecLSNs of the pages in the DirtyPageTable. If there are no dirty pages, it sets RedoLSN to the LSN of the checkpoint log record. The redo pass starts its scan of the log from RedoLSN. All the log records earlier than this point have already been applied to the database pages on disk. The analysis pass initially sets the list of transactions to be undone, undo-list, to the list of transactions in the checkpoint log record. The analysis pass also reads from the checkpoint log record the LSNs of the last log record for each transaction in undo-list.

The analysis pass continues scanning forward from the checkpoint. Whenever it finds a log record for a transaction not in the undo-list, it adds the transaction to undo-list. Whenever it finds a transaction end log record, it deletes the transaction from undo-list. All transactions left in undo-list at the end of analysis have to be rolled back later, in the undo pass. The analysis pass also keeps track of the last record of each transaction in undo-list, which is used in the undo pass.

The analysis pass also updates DirtyPageTable whenever it finds a log record for an update on a page. If the page is not in DirtyPageTable, the analysis pass adds it to DirtyPageTable, and sets the RecLSN of the page to the LSN of the log record.

#### 16.8.2.2 Redo Pass

The redo pass repeats history by replaying every action that is not already reflected in the page on disk. The redo pass scans the log forward from RedoLSN. Whenever it finds an update log record, it takes this action:

1. If the page is not in DirtyPageTable or the LSN of the update log record is less than the RecLSN of the page in DirtyPageTable, then the redo pass skips the log record.
2. Otherwise the redo pass fetches the page from disk, and if the PageLSN is less than the LSN of the log record, it redoes the log record.

Note that if either of the tests is negative, then the effects of the log record have already appeared on the page; otherwise the effects of the log record are not reflected on the page. Since ARIES allows non-idempotent physiological log records, a log record should not be redone if its effect is already reflected on the page. If the first test is negative, it is not even necessary to fetch the page from disk to check its PageLSN.

#### 16.8.2.3 Undo Pass and Transaction Rollback

The undo pass is relatively straightforward. It performs a single backward scan of the log, undoing all transactions in undo-list. The undo pass examines only log records of transactions in undo-list; the last LSN recorded during the analysis pass is used to find the last log record for each transaction in undo-list.

Whenever an update log record is found, it is used to perform an undo (whether for transaction rollback during normal processing, or during the restart undo pass). The undo pass generates a CLR containing the undo action performed (which must be physiological). It sets the UndoNextLSN of the CLR to the PrevLSN value of the update log record.

If a CLR is found, its UndoNextLSN value indicates the LSN of the next log record to be undone for that transaction; later log records for that transaction have already been rolled back. For log records other than CLRs, the PrevLSN field of the log record indicates the LSN of the next log record to be undone for that transaction. The next log record to be processed at each stop in the undo pass is the maximum, across all transactions in undo-list, of next log record LSN.

Figure 16.9 illustrates the recovery actions performed by ARIES, on an example log. We assume that the last completed checkpoint pointer on disk points to the checkpoint log record with LSN 7568. The PrevLSN values in the log records are shown using arrows in the figure, while the UndoNextLSN value is shown using a dashed arrow for the one compensation log record, with LSN 7565, in the figure. The analysis pass would start from LSN 7568, and when it is complete, RedoLSN would be 7564. Thus, the redo pass must start at the log record with LSN 7564. Note that this LSN is less than the LSN of the checkpoint log record, since the ARIES checkpointing algorithm does not flush modified pages to stable storage. The DirtyPageTable at the end of analysis would include pages 4894, 7200 from

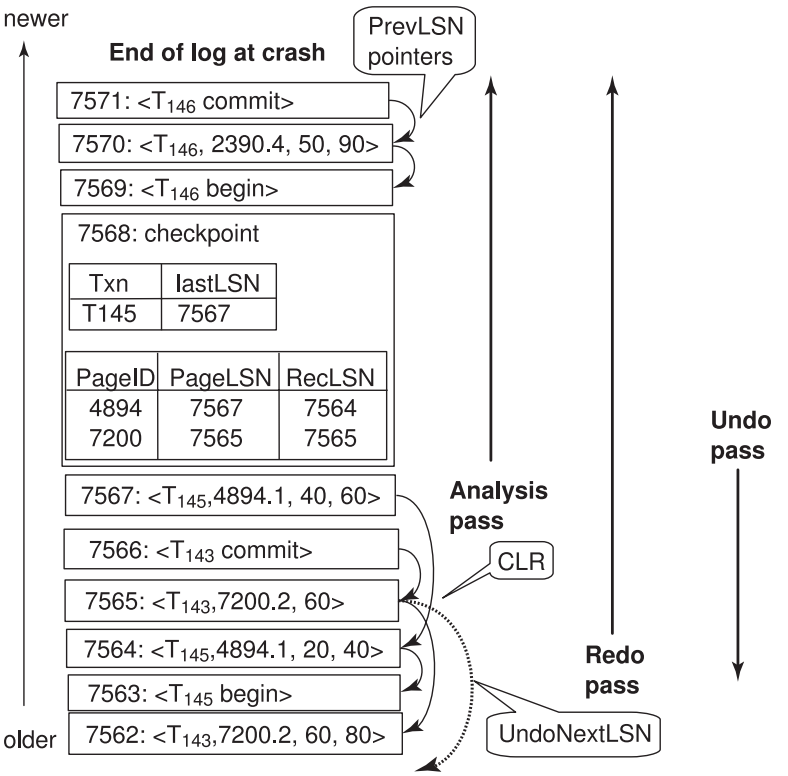



**Figure 16.9** Recovery actions in ARIES.

the checkpoint log record, and 2390 which is updated by the log record with LSN 7570. At the end of the analysis pass, the list of transactions to be undone consists of only $T_{145}$ in this example.

The redo pass for the above example starts from LSN 7564 and performs redo of log records whose pages appear in DirtyPageTable. The undo pass needs to undo only transaction $T_{145}$, and hence starts from its LastLSN value 7567, and continues backwards until the record $< T_{145}$ start> is found at LSN 7563.

### 16.8.3 Other Features

Among other key features that ARIES provides are:

- **Nested top actions**: ARIES allows the logging of operations that should not be undone even if a transaction gets rolled back; for example, if a transaction allocates a page to a relation, even if the transaction is rolled back the page allocation should not be undone since other transactions may have stored records in the page. Such operations that should not be undone are called nested top actions. Such operations can be modeled as operations whose undo action does nothing. In ARIES, such operations are implemented by creating a

dummy CLR whose UndoNextLSN is set such that transaction rollback skips the log records generated by the operation.

- **Recovery independence**: Some pages can be recovered independently from others, so that they can be used even while other pages are being recovered. If some pages of a disk fail, they can be recovered without stopping transaction processing on other pages.
- **Savepoints**: Transactions can record savepoints, and can be rolled back partially, up to a savepoint. This can be quite useful for deadlock handling, since transactions can be rolled back up to a point that permits release of required locks, and then restarted from that point.

  Programmers can also use savepoints to undo a transaction partially, and then continue execution; this approach can be useful to handle certain kinds of errors detected during the transaction execution.
- **Fine-grained locking**: The ARIES recovery algorithm can be used with index concurrency-control algorithms that permit tuple-level locking on indices, instead of page-level locking, which improves concurrency significantly.
- **Recovery optimizations**: The DirtyPageTable can be used to prefetch pages during redo, instead of fetching a page only when the system finds a log record to be applied to the page. Out-of-order redo is also possible: Redo can be postponed on a page being fetched from disk, and performed when the page is fetched. Meanwhile, other log records can continue to be processed.

In summary, the ARIES algorithm is a state-of-the-art recovery algorithm, incorporating a variety of optimizations designed to improve concurrency, reduce logging overhead, and reduce recovery time.

## 16.9 Remote Backup Systems

Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Increasingly, there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide **high availability**; that is, the time for which the system is unusable must be extremely small.

We can achieve high availability by performing transaction processing at one site, called the **primary site**, and having a **remote backup** site where all the data from the primary site are replicated. The remote backup site is sometimes also called the **secondary site**. The remote site must be kept synchronized with the primary site, as updates are performed at the primary. We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary—for example, we can locate it in a different state—so that a disaster at the primary does not damage

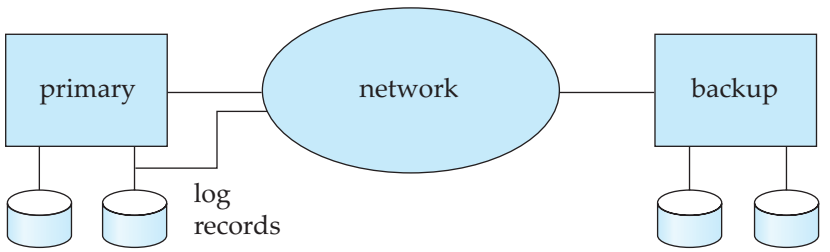



**Figure 16.10** Architecture of remote backup system.

the remote backup site. Figure 16.10 shows the architecture of a remote backup system.

When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its (perhaps outdated) copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered. Standard recovery algorithms, with minor modifications, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions.

Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost.

Several issues must be addressed in designing a remote backup system:

- **Detection of failure**. It is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup. For example, several independent network connections, including perhaps a modem connection over a telephone line, may be used. These connections may be backed up via manual intervention by operators, who can communicate over the telephone system.
- **Transfer of control**. When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down.

  The simplest way of transferring control is for the old primary to receive redo logs from the old backup site, and to catch up with the updates by applying them locally. The old primary can then act as a remote backup site. If control must be transferred back, the old backup site can pretend to have failed, resulting in the old primary taking over.

- **Time to recover**. If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received and can perform a checkpoint, so that earlier parts of the log can be deleted. The delay before the remote backup takes over can be significantly reduced as a result.

  A **hot-spare** configuration can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

- **Time to commit**. To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability. The degrees of durability can be classified as follows:

  - **One-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary site.

    The problem with this scheme is that the updates of a committed transaction may not have made it to the backup site, when the backup site takes over processing. Thus, the updates may appear to be lost. When the primary site recovers, the lost updates cannot be merged in directly, since the updates may conflict with later updates performed at the backup site. Thus, human intervention may be required to bring the database to a consistent state.

  - **Two-very-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site.

    The problem with this scheme is that transaction processing cannot proceed if either the primary or the backup site is down. Thus, availability is actually less than in the single-site case, although the probability of data loss is much less.

  - **Two-safe**. This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site.

    This scheme provides better availability than does two-very-safe, while avoiding the problem of lost transactions faced by the one-safe scheme. It results in a slower commit than the one-safe scheme, but the benefits generally outweigh the cost.

Several commercial shared-disk systems provide a level of fault tolerance that is intermediate between centralized and remote backup systems. In these commercial systems, the failure of a CPU does not result in system failure. Instead, other CPUs take over, and they carry out recovery. Recovery actions include rollback

of transactions running on the failed CPU, and recovery of locks held by those transactions. Since data are on a shared disk, there is no need for transfer of log records. However, we should safeguard the data from disk failure by using, for example, a RAID disk organization.

An alternative way of achieving high availability is to use a distributed database, with data replicated at more than one site. Transactions are then required to update all replicas of any data item that they update. We study distributed databases, including replication, in Chapter 19.

## 16.10 Summary

- A computer system, like any other mechanical or electrical device, is subject to failure. There are a variety of causes of such failure, including disk crash, power failure, and software errors. In each of these cases, information concerning the database system is lost.
- In addition to system failures, transactions may also fail for various reasons, such as violation of integrity constraints or deadlocks.
- An integral part of a database system is a recovery scheme that is responsible for the detection of failures and for the restoration of the database to a state that existed before the occurrence of the failure.
- The various types of storage in a computer are volatile storage, nonvolatile storage, and stable storage. Data in volatile storage, such as in RAM, are lost when the computer crashes. Data in nonvolatile storage, such as disk, are not lost when the computer crashes, but may occasionally be lost because of failures such as disk crashes. Data in stable storage are never lost.
- Stable storage that must be accessible online is approximated with mirrored disks, or other forms of RAID, which provide redundant data storage. Offline, or archival, stable storage may consist of multiple tape copies of data stored in a physically secure location.
- In case of failure, the state of the database system may no longer be consistent; that is, it may not reflect a state of the world that the database is supposed to capture. To preserve consistency, we require that each transaction be atomic. It is the responsibility of the recovery scheme to ensure the atomicity and durability property.
- In log-based schemes, all updates are recorded on a log, which must be kept in stable storage. A transaction is considered to have committed when its last log record, which is the commit log record for the transaction, has been output to stable storage.
- Log records contain old values and new values for all updated data items. The new values are used in case the updates need to be redone after a system crash. The old values are used to roll back the updates of the transaction if

the transaction aborts during normal operation, as well as to roll back the updates of the transaction in case the system crashed before the transaction committed.

- In the deferred-modifications scheme, during the execution of a transaction, all the write operations are deferred until the transaction has been committed, at which time the system uses the information on the log associated with the transaction in executing the deferred writes. With deferred modification, log records do not need to contain old values of updated data items.
- To reduce the overhead of searching the log and redoing transactions, we can use checkpointing techniques.
- Modern recovery algorithms are based on the concept of repeating history, whereby all actions taken during normal operation (since the last completed checkpoint) are replayed during the redo pass of recovery. Repeating history restores the system state to what it was at the time the last log record was output to stable storage before the system crashed. Undo is then performed from this state, by executing an undo pass that processes log records of incomplete transactions in reverse order.
- Undo of an incomplete transaction writes out special redo-only log records, and an abort log record. After that, the transaction can be considered to have completed, and it will not be undone again.
- Transaction processing is based on a storage model in which main memory holds a log buffer, a database buffer, and a system buffer. The system buffer holds pages of system object code and local work areas of transactions.
- Efficient implementation of a recovery scheme requires that the number of writes to the database and to stable storage be minimized. Log records may be kept in volatile log buffer initially, but must be written to stable storage when one of the following conditions occurs:
  - Before the $<T_i$ commit$>$ log record may be output to stable storage, all log records pertaining to transaction $T_i$ must have been output to stable storage.
  - Before a block of data in main memory is output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage.
- Modern recovery techniques support high-concurrency locking techniques, such as those used for $B^+$-tree concurrency control. These techniques allow early release of lower-level locks obtained by operations such as inserts or deletes, which allows other such operations to be performed by other transactions. After lower-level locks are released, physical undo is not possible, and instead logical undo, such as a deletion to undo an insertion, is required. Transactions retain higher-level locks that ensure that concurrent transac-

tions cannot perform actions that could make logical undo of an operation impossible.

- To recover from failures that result in the loss of nonvolatile storage, we must dump the entire contents of the database onto stable storage periodically—say, once per day. If a failure occurs that results in the loss of physical database blocks, we use the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, we use the log to bring the database system to the most recent consistent state.
- The ARIES recovery scheme is a state-of-the-art scheme that supports a number of features to provide greater concurrency, reduce logging overheads, and minimize recovery time. It is also based on repeating history, and allows logical undo operations. The scheme flushes pages on a continuous basis and does not need to flush all pages at the time of a checkpoint. It uses log sequence numbers (LSNs) to implement a variety of optimizations that reduce the time taken for recovery.
- Remote backup systems provide a high degree of availability, allowing transaction processing to continue even if the primary site is destroyed by a fire, flood, or earthquake. Data and log records from a primary site are continually backed up to a remote backup site. If the primary site fails, the remote backup site takes over transaction processing, after executing certain recovery actions.

## Review Terms

- Recovery scheme
- Failure classification
  - Transaction failure
  - Logical error
  - System error
  - System crash
  - Data-transfer failure
- Fail-stop assumption
- Disk failure
- Storage types
  - Volatile storage
  - Nonvolatile storage
  - Stable storage
- Blocks
  - Physical blocks
  - Buffer blocks
- Disk buffer
- Force-output
- Log-based recovery
- Log
- Log records
- Update log record
- Deferred modification
- Immediate modification
- Uncommitted modifications
- Checkpoints
- Recovery algorithm

- Restart recovery
- Transaction rollback
- Physical undo
- Physical logging
- Transaction rollback
- Checkpoints
- Restart recovery
- Redo phase
- Undo phase
- Repeating history
- Buffer management
- Log-record buffering
- Write-ahead logging (WAL)
- Log force
- Database buffering
- Latches
- Operating system and buffer management
- Fuzzy checkpointing
- Early lock release
- Logical operations
- Logical logging
- Logical undo
- Loss of nonvolatile storage
- Archival dump
- Fuzzy dump
- ARIES
  - Log sequence number (LSN)
  - PageLSN
  - Physiological redo
  - Compensation log record (CLR)
  - DirtyPageTable
  - Checkpoint log record
  - Analysis phase
  - Redo phase
  - Undo phase
- High availability
- Remote backup systems
  - Primary site
  - Remote backup site
  - Secondary site
- Detection of failure
- Transfer of control
- Time to recover
- Hot-spare configuration
- Time to commit
  - One-safe
  - Two-very-safe
  - Two-safe

## Practice Exercises

**16.1** Explain why log records for transactions on the undo-list must be processed in reverse order, whereas redo is performed in a forward direction.

**16.2** Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect:

- System performance when no failure occurs?
- The time it takes to recover from a system crash?
- The time it takes to recover from a media (disk) failure?

**16.3** Some database systems allow the administrator to choose between two forms of logging: *normal logging*, used to recover from system crashes, and *archival logging*, used to recover from media (disk) failure. When can a log record be deleted, in each of these cases, using the recovery algorithm of Section 16.4?

**16.4** Describe how to modify the recovery algorithm of Section 16.4 to implement savepoints, and to perform rollback to a savepoint. (Savepoints are described in Section 16.8.3.)

**16.5** Suppose the deferred modification technique is used in a database.

a. Is the old-value part of an update log record required any more? Why or why not?

b. If old values are not stored in update log records, transaction undo is clearly not feasible. How would the redo-phase of recovery have to be modified as a result?

c. Deferred modification can be implemented by keeping updated data items in local memory of transactions, and reading data items that have not been updated directly from the database buffer. Suggest how to efficiently implement a data item read, ensuring that a transaction sees its own updates.

d. What problem would arise with the above technique, if transactions perform a large number of updates?

**16.6** The shadow-paging scheme requires the page table to be copied. Suppose the page table is represented as a $B^+$-tree.

a. Suggest how to share as many nodes as possible between the new copy and the shadow-copy of the $B^+$-tree, assuming that updates are made only to leaf entries, with no insertions and deletions.

b. Even with the above optimization, logging is much cheaper than a shadow-copy scheme, for transactions that perform small updates. Explain why.

**16.7** Suppose we (incorrectly) modify the recovery algorithm of Section 16.4 to not log actions taken during transaction rollback. When recovering from a system crash, transactions that were rolled back earlier would then be included in undo-list, and rolled back again. Give an example to show how actions taken during the undo phase of recovery could result in an incorrect database state. (Hint: Consider a data item updated by an aborted transaction, and then updated by a transaction that commits.)

**16.8** Disk space allocated to a file as a result of a transaction should not be released even if the transaction is rolled back. Explain why, and explain how ARIES ensures that such actions are not rolled back.

**16.9** Suppose a transaction deletes a record, and the free space generated thus is allocated to a record inserted by another transaction, even before the first transaction commits.

a. What problem can occur if the first transaction needs to be rolled back?

b. Would this problem be an issue if page-level locking is used instead of tuple-level locking?

c. Suggest how to solve this problem while supporting tuple-level locking, by logging post-commit actions in special log records, and executing them after commit. Make sure your scheme ensures that such actions are performed exactly once.

**16.10** Explain the reasons why recovery of interactive transactions is more difficult to deal with than is recovery of batch transactions. Is there a simple way to deal with this difficulty? (Hint: Consider an automatic teller machine transaction in which cash is withdrawn.)

**16.11** Sometimes a transaction has to be undone after it has committed because it was erroneously executed, for example because of erroneous input by a bank teller.

a. Give an example to show that using the normal transaction undo mechanism to undo such a transaction could lead to an inconsistent state.

b. One way to handle this situation is to bring the whole database to a state prior to the commit of the erroneous transaction (called *point-in-time* recovery). Transactions that committed later have their effects rolled back with this scheme.

Suggest a modification to the recovery algorithm of Section 16.4 to implement point-in-time recovery using database dumps.

c. Later nonerroneous transactions can be re-executed logically, if the updates are available in the form of SQL but cannot be re-executed using their log records. Why?

## Exercises

**16.12** Explain the difference between the three storage types—volatile, nonvolatile, and stable—in terms of I/O cost.

**16.13** Stable storage cannot be implemented.

a. Explain why it cannot be.

b. Explain how database systems deal with this problem.