PART 4

Data Storage and Querying

Although a database system provides a high-level view of data, ultimately data have to be stored as bits on one or more storage devices. A vast majority of databases today store data on magnetic disk and fetch data into main space memory for processing, or copy data onto tapes and other backup devices for archival storage. The physical characteristics of storage devices play a major role in the way data are stored, in particular because access to a random piece of data on disk is much slower than memory access: Disk access takes tens of milliseconds, whereas memory access takes a tenth of a microsecond.

Chapter 11 begins with an overview of physical storage media, including mechanisms to minimize the chance of data loss due to failures. The chapter then describes how records are mapped to files, which in turn are mapped to bits on the disk. Storage and retrieval of objects is also covered in Chapter 11.

Many queries reference only a small proportion of the records in a file. An index is a structure that helps locate desired records of a relation quickly, without examining all records. The index in this textbook is an example, although, unlike database indices, it is meant for human use. Chapter 12 describes several types of indices used in database systems.

User queries have to be executed on the database contents, which reside on storage devices. It is usually convenient to break up queries into smaller operations, roughly corresponding to the relational algebra operations. Chapter 13 describes how queries are processed, presenting algorithms for implementing individual operations, and then outlining how the operations are executed in synchrony, to process a query.

There are many alternative ways of processing a query, which can have widely varying costs. Query optimization refers to the process of finding the lowest-cost method of evaluating a given query. Chapter 14 describes the process of query optimization.

CHAPTER 11

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 12, 13, and 14, 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 will 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.

11.1 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 small; its use is managed by the computer system hardware. We shall not be concerned about managing cache storage in the database system.
- **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 many megabytes of

data, or even 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**. Also known as *electrically erasable programmable read-only memory* (*EEPROM*), flash memory differs from main memory in that data survive power failure. Reading data from flash memory takes less than 100 nanoseconds (a nanosecond is 1/1000 of a microsecond), which is roughly as fast as reading data from main memory. However, writing data to flash memory is more complicated—data can be written once, which takes about 4 to 10 microseconds, but cannot be overwritten directly. To overwrite memory that has been written already, we have to erase an entire bank of memory at once; it is then ready to be written again. A drawback of flash memory is that it can support only a limited number of erase cycles, ranging from 10,000 to 1 million. Flash memory has found popularity as a replacement for magnetic disks for storing small volumes of data (5 to 10 megabytes) in low-cost computer systems, such as computer systems that are embedded in other devices, in hand-held computers, and in other digital electronic devices such as digital cameras.

- **Magnetic-disk storage**. The primary medium for the long-term on-line 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.

  The size of magnetic disks currently ranges from a few gigabytes to 80 gigabytes. Both the lower and upper end of this range have been growing at about 50 percent per year, and we can expect much larger capacity disks 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 640 megabytes of data, 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). Data are stored optically on a disk, and are read by a laser. The optical disks used in read-only compact disks (CD-ROM) or read-only digital video disk (DVD-ROM) cannot be written, but are supplied with data prerecorded.

  There are "record-once" versions of compact disk (called CD-R) and digital video disk (called 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 and DVD-RAM), which can be written multiple times. Recordable compact disks are magnetic–optical storage devices that use optical means to read magnetically encoded data. Such disks are useful for archival storage of data as well as distribution of data.

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 much 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 gigabyte to 300 gigabytes tapes are currently available), and can be removed from the tape drive, so they are well suited to cheap archival storage. Tape jukeboxes are used to hold exceptionally large collections of data, such as remote-sensing data from satellites, which could include as much as hundreds of terabytes (1 terabyte = 10^{12} bytes), or even a petabyte (1 petabyte = 10^{15} bytes) of data.

The various storage media can be organized in a hierarchy (Figure 11.1) 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

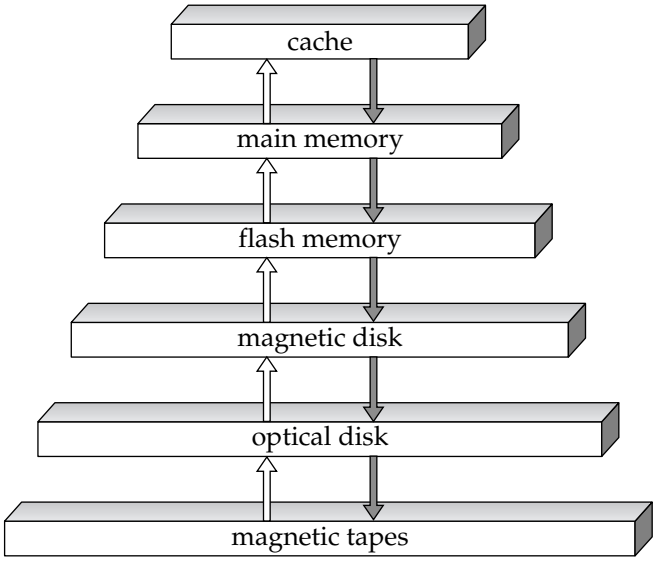


Figure 11.1 Storage-device hierarchy.

storage capacity. Today, almost all active data are stored on disks, except in 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 11.1, the storage systems from main memory up are volatile, whereas the storage systems below main memory are nonvolatile. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. We shall return to this subject in Chapter 17.

11.2 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Disk capacities have been growing at over 50 percent per year, but 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 large database may require hundreds of disks.

11.2.1 Physical Characteristics of Disks

Physically, disks are relatively simple (Figure 11.2). 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 and are covered (usually on both sides) with magnetic recording material. We call such magnetic disks **hard disks**, to distinguish them from **floppy disks**, which are made from flexible material.

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 typically 512 bytes; there are over 16,000 tracks on each platter, and 2 to 4 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 200 sectors per track in the inner tracks, and around 400 sectors per track in the outer tracks. The numbers above 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. There may be hundreds of concentric tracks on a disk surface, containing thousands of sectors.

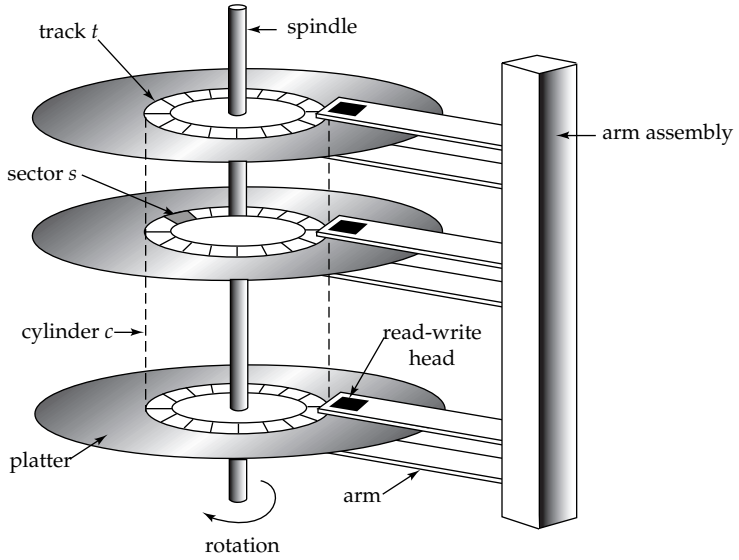


Figure 11.2 Moving-head disk mechanism.

Each side of a platter of a disk has a read–write head, which 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 ith track, the heads on all other platters are also on the ith track of their respective platters. Hence, the ith tracks of all the platters together are called the ith **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. Smaller-diameter disks are used in portable devices such as laptop computers.

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. Usually, the head touching the surface causes the removed medium to become airborne and to come between the other heads and their platters, causing more crashes. Under normal circumstances, a head crash results in failure of the entire disk, which must then be replaced. 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 *fixed-head disk* has a separate head for each track. This arrangement allows the computer to switch from track to track quickly, without having to move the head assembly, but because of the large number of heads, the device is extremely expensive. Some disk systems have multiple disk arms, allowing more than one track on the same platter to be accessed at a time. Fixed-head disks and multiple-arm disks were used in high-performance mainframe systems, but are no longer in production.

A **disk controller** interfaces between the computer system and the actual hardware of the disk drive. It 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.

Figure 11.3 shows how disks are connected to a computer system. Like other storage units, disks are connected to a computer system or to a controller through a high-speed interconnection. In modern disk systems, lower-level functions of the disk controller, such as control of the disk arm, computing and verification of checksums, and remapping of bad sectors, are implemented within the disk drive unit.

The **AT attachment** (**ATA**) interface (which is a faster version of the **integrated drive electronics** (IDE) interface used earlier in IBM PCs) and a **small-computer-system interconnect** (SCSI; pronounced "scuzzy") are commonly used to connect

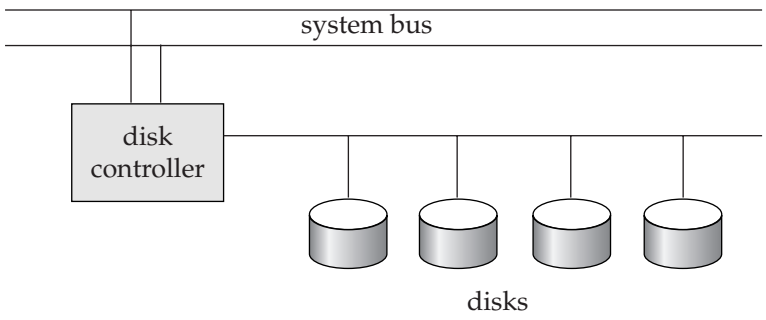


Figure 11.3 Disk subsystem.

disks to personal computers and workstations. Mainframe and server systems usually have a faster and more expensive interface, such as high-capacity versions of the SCSI interface, and the Fibre Channel interface.

While disks are usually connected directly by cables to the disk controller, 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 of server computers. The disks are usually organized locally using **redundant arrays of independent disks** (**RAID**) storage organizations, but the RAID organization may be hidden from the server computers: the disk subsystems pretend each RAID system is a very large and very reliable disk. The controller and the disk continue to use SCSI or Fibre Channel interfaces 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, which 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.

11.2.2 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 milliseconds and 10 milliseconds, depending on the disk model.

Once the seek has started, 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 claim to support maximum transfer rates of about 25 to 40 megabytes per second, although actual transfer rates may be significantly less, at about 4 to 8 megabytes per second.

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 30,000 to 1,200,000 hours—about 3.4 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.

There may be multiple disks sharing a disk interface. The widely used ATA-4 interface standard (also called Ultra-DMA) supports 33 megabytes per second transfer rates, while ATA-5 supports 66 megabytes per second. SCSI-3 (Ultra2 wide SCSI) supports 40 megabytes per second, while the more expensive Fibre Channel interface supports up to 256 megabytes per second. The transfer rate of the interface is shared between all disks attached to the interface.

11.2.3 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 contiguous sequence of sectors from a single track of one platter. Block sizes range from 512 bytes to several kilobytes. Data are transferred between disk and main memory in units of blocks. The lower levels of the file-system manager convert block addresses into the hardware-level cylinder, surface, and sector number.

Since access to data on disk is several orders of magnitude slower than access to data in main memory, equipment designers have focused on techniques for improving the speed of access to blocks on disk. One such technique, buffering of blocks in memory to satisfy future requests, is discussed in Section 11.5. Here, we discuss several other techniques.

- **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 algorithms 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. Now, 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 personal-computer operating systems, hide the disk organization from users, and manage the allocation internally. However, over time, a sequential file 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, but the system is generally unusable while these utilities operate.

- **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** (NV-RAM) to speed up disk writes drastically. The contents of nonvolatile RAM are not lost in power failure. A common way to implement nonvolatile RAM is to use battery–backed-up RAM. 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 a nonvolatile RAM 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 nonvolatile RAM buffer becomes full. When the database system requests a block write, it notices a delay only if the nonvolatile RAM buffer

is full. On recovery from a system crash, any pending buffered writes in the nonvolatile RAM are written back to the disk.

An example illustrates how much nonvolatile RAM improves performance. Assume that write requests are received in a random fashion, with the disk being busy on average 90 percent of the time.[1] If we have a nonvolatile RAM buffer of 50 blocks, then, on average, only once per minute will a write find the buffer to be full (and therefore have to wait for a disk write to finish). Doubling the buffer to 100 blocks results in approximately only one write per hour finding the buffer to be full. Thus, in most cases, disk writes can be executed without the database system waiting for a seek or rotational latency.

- **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.

  The **log-based file system** is an extreme version of the log-disk approach. Data are not written back to their original destination on disk; instead, the file system keeps track of where in the log disk the blocks were written most recently, and retrieves them from that location. The log disk itself is compacted periodically, so that old writes that have subsequently been overwritten can be removed. This approach improves write performance, but generates a high degree of fragmentation for files that are updated often. As we noted earlier, such fragmentation increases seek time for sequential reading of files.

11.3 RAID

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

1. For the statistically inclined reader, we assume Poisson distribution of arrivals. The exact arrival rate and rate of service are not needed since the disk utilization provides enough information for our calculations.

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. Parallelism can also be used to perform several independent reads or writes 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.

11.3.1 Improvement of Reliability via Redundancy

Let us first consider reliability. The chance that some 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 11.2.1). Such a high rate 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, then the **mean time to data loss** of a mirrored disk system is $100000^2/(2 * 10) = 500 * 10^6$ 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 Exercise 11.4.

11.3.2 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 $(i \bmod n) + 1$; it uses the $\lfloor i/n \rfloor$th physical 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.

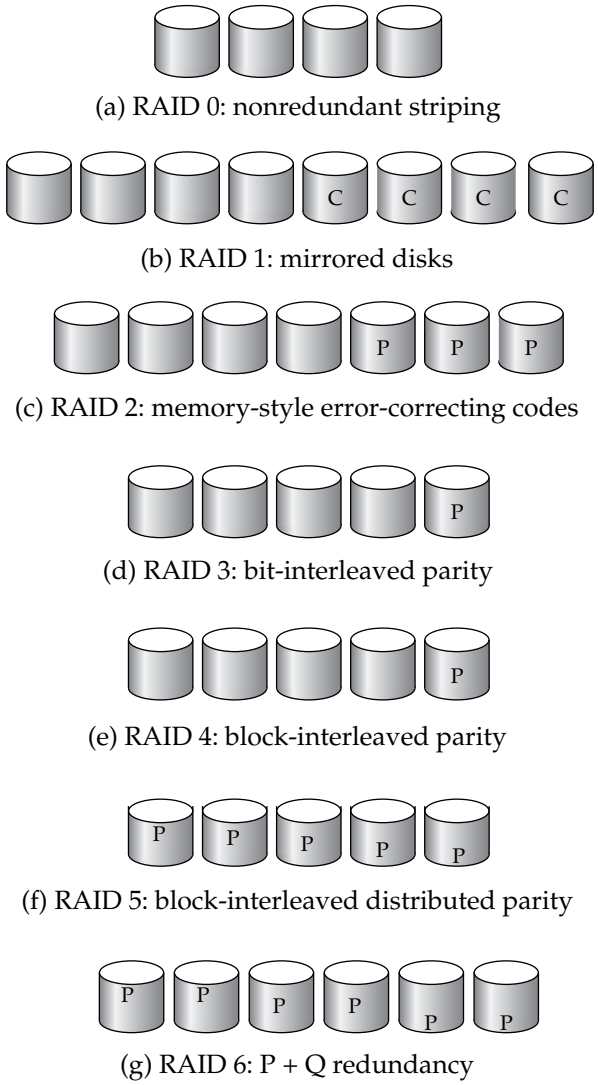
11.3.3 RAID Levels

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 11.4. (In the figure, P indicates error-correcting bits, and C indicates a second copy of the data.) For all levels, the figure depicts four disk's 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 11.4a shows an array of size 4.
- **RAID level 1** refers to disk mirroring with block striping. Figure 11.4b shows a mirrored organization that holds four disks worth of data.
- **RAID level 2**, known as memory-style error-correcting-code (ECC) organization, employs parity bits. Memory systems have long used parity bits for error detection and correction. Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte that are set to 1 is even (parity = 0) or odd (parity = 1). If one of the bits in the byte gets damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit gets damaged, it will not match the computed parity. Thus, all 1-bit errors will be detected by the memory system. Error-correcting schemes store 2 or more extra bits, and can reconstruct the data if a single 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 1, the second bit in disk 2, and so on until the eighth bit is stored in disk 8, and the error-correction bits are stored in further disks.

  Figure 11.4c 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 11.4c 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.



(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 11.4 RAID levels.

- **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 11.4d 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 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 11.4e. 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 11.4f shows the setup. The P's are distributed across all the disks. For example, with an array of 5 disks, the parity block, labelled Pk, for logical blocks $4k, 4k+1, 4k+2, 4k+3$ is stored in disk $(k \bmod 5) + 1$; 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 11.4g, 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.

Some vendors use their own terminology to describe their RAID implementations.[2] However, the terminology we have presented is the most widely used.

11.3.4 Choice of RAID Level

The factors to be taken into account when 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 is being rebuilt on a new disk)

The time to rebuild the data of a failed disk can be significant, and 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.

2. For example, some products use RAID level 1 to refer to mirroring without striping, and level 1+0 or level 10 to refer to mirroring with striping. Such a distinction is not really necessary since not striping can simply be viewed as a special case of striping, namely striping across 1 disk.

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 rarely used since block striping (level 5) 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 would 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.

11.3.5 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 that need to be executed; in case of power failure before a write is completed, when the system comes back up, it retrieves information about 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 Exercise 11.4).

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 controllers, and multiple interconnections to connect them 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.

11.3.6 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.

11.4 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.

11.4.1 Optical Disks

Compact disks are a popular medium for distributing software, multimedia data such as audio and images, and other electronically published information. They have a fairly large capacity (640 megabytes), and they are cheap to mass-produce.

Digital video disks (DVDs) are replacing compact disks in applications that require very large 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.

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 15 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 $12\times$ for DVD are now common.

The record-once versions of optical disks (CD-R, and increasingly, 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, 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 (usually, 1 to 10) of drives. 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 slower than disk access times.

11.4.2 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 offline 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.

Some tape formats (such as the **Accelis** format) support faster seek times (of the order of tens of seconds), which is important for applications that need quick access to very large amounts of data, larger than what would fit economically on a disk drive. 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 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 terabytes (10^{12} 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 by remote-sensing satellites, and large video libraries for television broadcasters.

11.5 Storage Access

A database is mapped into a number of different files, which are maintained by the underlying operating system. These files reside permanently on disks, with backups on tapes. Each file is partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer. We shall discuss in Section 11.6 various ways to organize the data logically in files.

A block may contain several data items. The exact set of data items that a block contains is determined by the form of physical data organization being used (see Section 11.6). We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as our banking example.

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**.

11.5.1 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 may be much more 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 17), 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 17; briefly, main-memory contents and thus buffer contents are lost in a crash, whereas data on disk usually survive a crash.

11.5.2 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

```
for each tuple b of borrower do
    for each tuple c of customer do
        if b[customer-name] = c[customer-name]
        then begin
                let x be a tuple defined as follows:
                x[customer-name] := b[customer-name]
                x[loan-number] := b[loan-number]
                x[customer-street] := c[customer-street]
                x[customer-city] := c[customer-city]
                include tuple x as part of result of borrower ⋈ customer
             end
    end
end
```

Figure 11.5 Procedure for computing join.

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.

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 relational-algebra expression

$$borrower \bowtie customer$$

Assume that the strategy chosen to process this request is given by the pseudocode program shown in Figure 11.5. (We shall study other strategies in Chapter 13.)

Assume that the two relations of this example are stored in separate files. In this example, we can see that, once a tuple of *borrower* has been processed, that tuple is not needed again. Therefore, once processing of an entire block of *borrower* 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 a *borrower* block as soon as the final tuple has been processed. This buffer-management strategy is called the **toss-immediate** strategy.

Now consider blocks containing *customer* tuples. We need to examine every block of *customer* tuples once for each tuple of the *borrower* relation. When processing of a *customer* block is completed, we know that that block will not be accessed again until all other *customer* blocks have been processed. Thus, the most recently used *customer* block will be the final block to be re-referenced, and the least recently used

customer 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 is the **most recently used** (**MRU**) strategy. If a *customer* block must be removed from the buffer, the MRU strategy chooses the most recently used block.

For the MRU strategy to work correctly for our example, the system must pin the *customer* block currently being processed. After the final *customer* 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 11.8) 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 12, 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 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 16) 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 17) 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 17, we define precisely the interaction between the buffer manager and the crash-recovery subsystem.

11.6 File Organization

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

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Figure 11.6 File containing *account* records.

assume the existence of an underlying *file system*. We need to consider ways of representing logical data models in terms of files.

Although blocks are of a fixed size determined by the physical properties of the disk and by the operating system, record sizes vary. 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.

11.6.1 Fixed-Length Records

As an example, let us consider a file of *account* records for our bank database. Each record of this file is defined as:

```
type deposit = record
                   account-number : char(10);
                   branch-name : char (22);
                   balance : real;
               end
```

If we assume that each character occupies 1 byte and that a real occupies 8 bytes, our *account* record is 40 bytes long. A simple approach is to use the first 40 bytes for the first record, the next 40 bytes for the second record, and so on (Figure 11.6). However, there are two problems with this simple approach:

1. 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	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Figure 11.7 File of Figure 11.6, with record 2 deleted and all records moved.

2. Unless the block size happens to be a multiple of 40 (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.

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 11.7). 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 11.8).

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

At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted. We use this

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600

Figure 11.8 File of Figure 11.6, with record 2 deleted and final record moved.

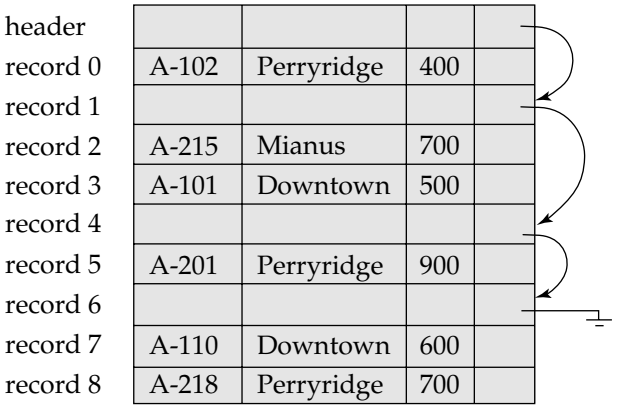


Figure 11.9 File of Figure 11.6, with free list after deletion of records 1, 4, and 6.

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 11.9 shows the file of Figure 11.6, 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.

11.6.2 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

Different techniques for implementing variable-length records exist. For purposes of illustration, we shall use one example to demonstrate the various implementation techniques. We shall consider a different representation of the *account* information stored in the file of Figure 11.6, in which we use one variable-length record for each branch name and for all the account information for that branch. The format of the record is

```
type account-list = record
                        branch-name : char (22);
                        account-info : array [1 .. ∞] of
                                           record;
                                               account-number : char(10);
                                               balance : real;
                                           end
                    end
```

We define *account-info* as an array with an arbitrary number of elements. That is, the type definition does not limit the number of elements in the array, although any actual record will have a specific number of elements in its array. There is no limit on how large a record can be (up to, of course, the size of the disk storage!).

11.6.2.1 Byte-String Representation

A simple method for implementing variable-length records is to attach a special *end-of-record* (⊥) symbol to the end of each record. We can then store each record as a string of consecutive bytes. Figure 11.10 shows such an organization to represent the file of fixed-length records of Figure 11.6 as variable-length records. An alternative version of the byte-string representation stores the record length at the beginning of each record, instead of using end-of-record symbols.

The byte-string representation as described in Figure 11.10 has some disadvantages:

- It is not easy to reuse space occupied formerly by a deleted record. Although techniques exist to manage insertion and deletion, they lead to a large number of small fragments of disk storage that are wasted.
- There is no space, in general, for records to grow longer. If a variable-length record becomes longer, it must be moved—movement is costly if pointers to the record are stored elsewhere in the database (e.g., in indices, or in other records), since the pointers must be located and updated.

Thus, the basic byte-string representation described here not usually used for implementing variable-length records. However, a modified form of the byte-string repre-

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

Figure 11.10 Byte-string representation of variable-length records.

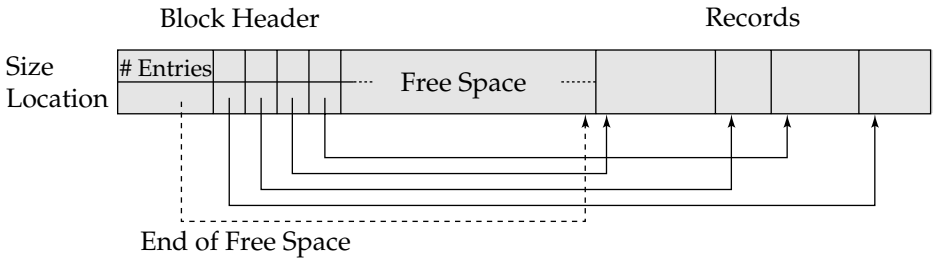


Figure 11.11 Slotted-page structure.

sentation, called the slotted-page structure, is commonly used for organizing records *within* a single block.

The **slotted-page structure** appears in Figure 11.11. 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: A typical value is 4 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 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.

11.6.2.2 Fixed-Length Representation

Another way to implement variable-length records efficiently in a file system is to use one or more fixed-length records to represent one variable-length record.

There are two ways of doing this:

1. **Reserved space**. If there is a maximum record length that is never exceeded, we can use fixed-length records of that length. Unused space (for records

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

Figure 11.12 File of Figure 11.10, using the reserved-space method.

shorter than the maximum space) is filled with a special null, or end-of-record, symbol.

2. **List representation**. We can represent variable-length records by lists of fixed-length records, chained together by pointers.

If we choose to apply the reserved-space method to our account example, we need to select a maximum record length. Figure 11.12 shows how the file of Figure 11.10 would be represented if we allowed a maximum of three accounts per branch. A record in this file is of the *account-list* type, but with the array containing exactly three elements. Those branches with fewer than three accounts (for example, Round Hill) have records with null fields. We use the symbol ⊥ to represent this situation in Figure 11.12. In practice, a particular value that can never represent real data is used (for example, an account number that is blank, or a name beginning with "*").

The reserved-space method is useful when most records have a length close to the maximum. Otherwise, a significant amount of space may be wasted. In our bank example, some branches may have many more accounts than others. This situation leads us to consider the linked list method. To represent the file by the linked list method, we add a pointer field as we did in Figure 11.9. The resulting structure appears in Figure 11.13.

Figure 11.13 File of Figure 11.10 using linked lists.

Figure 11.14 Anchor-block and overflow-block structures.

The file structures of Figures 11.9 and 11.13 both use pointers; the difference is that, in Figure 11.9, we use pointers to chain together only deleted records, whereas in Figure 11.13, we chain together all records pertaining to the same branch.

A disadvantage to the structure of Figure 11.13 is that we waste space in all records except the first in a chain. The first record needs to have the *branch-name* value, but subsequent records do not. Nevertheless, we need to include a field for *branch-name* in all records, lest the records not be of fixed length. This wasted space is significant, since we expect, in practice, that each branch has a large number of accounts. To deal with this problem, we allow two kinds of blocks in our file:

1. **Anchor block**, which contains the first record of a chain
2. **Overflow block**, which contains records other than those that are the first record of a chain

Thus, all records *within a block* have the same length, even though not all records in the file have the same length. Figure 11.14 shows this file structure.

11.7 Organization of Records in Files

So far, we have studied how records are represented in a file structure. An instance of 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 11.7.1 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 file the record should be placed. Chapter 12 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 **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 11.7.2 describes this organization.

11.7.1 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 11.15 shows a sequential file of *account* records taken from our banking example. In that example, the records are stored in search-key order, using *branch-name* 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 13.

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 11.15 Sequential file for *account* records.

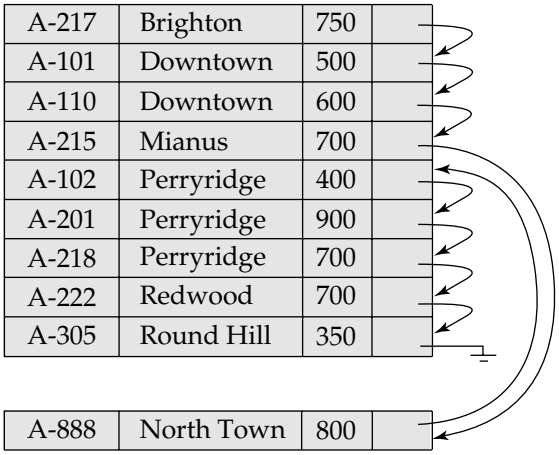


Figure 11.16 Sequential file after an insertion.

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 11.16 shows the file of Figure 11.15 after the insertion of the record (North Town, A-888, 800). The structure in Figure 11.16 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, 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 11.15 is not needed.

11.7.2 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. To see the advantage of storing many relations in one file, consider the following SQL query for the bank database:

select *account-number, customer-name, customer-street, customer-city*
from *depositor, customer*
where *depositor.customer-name* = *customer.customer-name*

This query computes a join of the *depositor* and *customer* relations. Thus, for each tuple of *depositor*, the system must locate the *customer* tuples with the same value for *customer-name*. Ideally, these records will be located with the help of *indices*, which we shall discuss in Chapter 12. 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.

As a concrete example, consider the *depositor* and *customer* relations of Figures 11.17 and 11.18, respectively. In Figure 11.19, we show a file structure designed for efficient execution of queries involving $depositor \bowtie customer$. The *depositor* tuples for each *customer-name* are stored near the *customer* tuple for the corresponding *customer-name*. This structure mixes together tuples of two relations, but allows for efficient processing of the join. When a tuple of the *customer* relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding

customer-name	*account-number*
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

Figure 11.17 The *depositor* relation.

customer-name	*customer-street*	*customer-city*
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

Figure 11.18 The *customer* relation.

depositor tuples are stored on the disk near the *customer* tuple, the block containing the *customer* tuple contains tuples of the *depositor* relation needed to process the query. If a customer has so many accounts that the *depositor* records do not fit in one block, the remaining records appear on nearby blocks.

A **clustering file organization** is a file organization, such as that illustrated in Figure 11.19 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.

Our use of clustering has enhanced processing of a particular join (*depositor* ⋈ *customer*), but it results in slowing processing of other types of query. For example,

select *
from *customer*

requires more block accesses than it did in the scheme under which we stored each relation in a separate file. Instead of several *customer* records appearing in one block, each record is located in a distinct block. Indeed, simply finding all the *customer* records is not possible without some additional structure. To locate all tuples of the *customer* relation in the structure of Figure 11.19, we need to chain together all the records of that relation using pointers, as in Figure 11.20.

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

11.8 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

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

Figure 11.19 Clustering file structure.

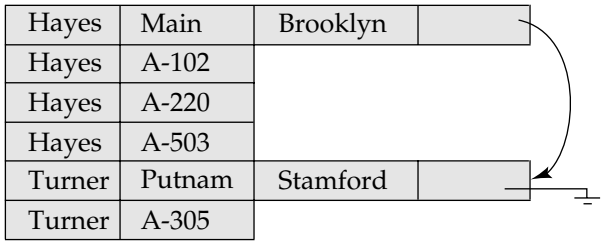


Figure 11.20 Clustering file structure with pointer chains.

schema of the relations. This information is 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
- 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 12, 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 information constitutes, in effect, a miniature database. Some database systems store this information by using special-purpose data structures and code. It is generally preferable to store the data about the database in the database itself. By using the database to store system data, 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 data by relations must be made by the system designers. One possible representation, with primary keys underlined, is

Relation-metadata (*relation-name*, *number-of-attributes*, *storage-organization*, *location*)
Attribute-metadata (*attribute-name*, *relation-name*, *domain-type*, *position*, *length*)
User-metadata (*user-name*, *encrypted-password*, *group*)
Index-metadata (*index-name*, *relation-name*, *index-type*, *index-attributes*)
View-metadata (*view-name*, *definition*)

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 "*branch-name, branch-city*". 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 non-normalized form to achieve fast access.

The storage organization and location of the *Relation-metadata* itself must be recorded elsewhere (for example, in the database code itself), since we need this information to find the contents of *Relation-metadata*.

11.9 Storage for Object-Oriented Databases**

The file-organization techniques described in Section 11.7—the heap, sequential, hashing and clustering organizations—can also be used for storing objects in an object-oriented database. However, some extra features are needed to support object-oriented database features, such as set-valued fields and persistent pointers.

11.9.1 Mapping of Objects to Files

The mapping of objects to files is in many ways like the mapping of tuples to files in a relational system. At the lowest level of data representation, both tuples and the data parts of objects are simply sequences of bytes. We can therefore store object data in the file structures described in this chapter, with some modifications which we note next.

Objects in object-oriented databases may lack the uniformity of tuples in relational databases. For example, fields of records may be sets; in relational databases, in con-

trast, data are typically required to be (at least) in first normal form. Furthermore, objects may be extremely large. Such objects have to be managed differently from records in a relational system.

We can implement set-valued fields that have a small number of elements using data structures such as linked lists. Set-valued fields that have a larger number of elements can be implemented as relations in the database. Set-valued fields of objects can also be eliminated at the *storage level* by normalization: A relation is created containing one tuple for each value of a set-valued field of an object. Each tuple also contains the object identifier of the object. However, this relation is not made visible to the upper levels of the database system. The storage system gives the upper levels of the database system the view of a set-valued field, even though the set-valued field has actually been normalized by creating a new relation.

Some applications include extremely large objects that are not easily decomposed into smaller components. Such large objects may each be stored in a separate file. We discuss this idea further in Section 11.9.6.

11.9.2 Implementation of Object Identifiers

Since objects are identified by object identifiers (OIDs), an object-storage system needs a mechanism to locate an object, given an OID. If the OIDs are **logical OID**s—that is, they do not specify the location of the object—then the storage system must maintain an index that maps OIDs to the actual location of the object. If the OIDs are **physical OID**s—that is, they encode the location of the object—then the object can be found directly. Physical OIDs typically have the following three parts:

1. A volume or file identifier
2. A block identifier within the volume or file
3. An offset within the block

A volume is a logical unit of storage that usually corresponds to a disk.

In addition, physical OIDs may contain a **unique identifier**, which is an integer that distinguishes the OID from the identifiers of other objects that happened to be stored at the same location earlier, and were deleted or moved elsewhere. The unique identifier is also stored with the object, and the identifiers in an OID and the corresponding object should match. If the unique identifier in a physical OID does not match the unique identifier in the object to which that OID points, the system detects that the pointer is a dangling pointer, and signals an error. (A **dangling pointer** is a pointer that does not point to a valid object.) Figure 11.21 illustrates this scheme.

Such pointer errors occur when physical OIDs corresponding to old objects that have been deleted are used accidentally. If the space occupied by the object had been reallocated, there may be a new object in the location, and it may get incorrectly addressed by the identifier of the old object. If a dangling pointer is not detected, it could cause corruption of a new object stored at the same location. The unique identifier helps to detect such errors, since the unique identifiers of the old physical OID and the new object will not match.

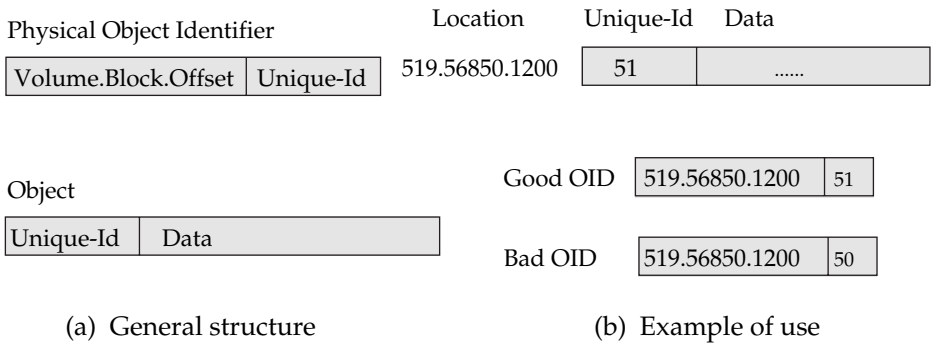


Figure 11.21 Unique identifiers in an OID.

Suppose that an object has to be moved to a new block, perhaps because the size of the object has increased, and the old block has no extra space. Then, the physical OID will point to the old block, which no longer contains the object. Rather than change the OID of the object (which involves changing every object that points to this one), we leave behind a **forwarding address** at the old location. When the database tries to locate the object, it finds the forwarding address instead of the object; it then uses the forwarding address to locate the object.

11.9.3 Management of Persistent Pointers

We implement persistent pointers in a persistent programming language by using OIDs. In some implementations, persistent pointers are physical OIDs; in others, they are logical OIDs. An important difference between persistent pointers and in-memory pointers is the size of the pointer. In-memory pointers need to be only big enough to address all virtual memory. On most current computers, in-memory pointers are usually 4 bytes long, which is sufficient to address 4 gigabytes of memory. The most recent computer architectures have pointers that are 8 bytes long,

Persistent pointers need to address all the data in a database. Since database systems are often bigger than 4 gigabytes, persistent pointers are usually at least 8 bytes long. Many object-oriented databases also provide unique identifiers in persistent pointers, to catch dangling references. This feature further increases the size of persistent pointers. Thus, persistent pointers may be substantially longer than in-memory pointers.

The action of looking up an object, given its identifier, is called **dereferencing**. Given an in-memory pointer (as in C++), looking up the object is merely a memory reference. Given a persistent pointer, dereferencing an object has an extra step—finding the actual location of the object in memory by looking up the persistent pointer in a table. If the object is not already in memory, it has to be loaded from disk. We can implement the table lookup fairly efficiently by using a hash table data structure, but the lookup is still slow compared to a pointer dereference, even if the object is already in memory.

Pointer swizzling is a way to cut down the cost of locating persistent objects that are already present in memory. The idea is that, when a persistent pointer is first dereferenced, the system locates the object and brings it into memory if it is not already there. Now the system carries out an extra step—it stores an in-memory pointer to the object in place of the persistent pointer. The next time that the *same* persistent pointer is dereferenced, the in-memory location can be read out directly, so the costs of locating the object are avoided. (When persistent objects have to be moved from memory back to disk to make space for other persistent objects, the system must carry out an extra step to ensure that the object is still in memory. Correspondingly, when an object is written out, any persistent pointers that it contained and that were swizzled have to be **deswizzled**, that is, converted back to their persistent representation. Pointer swizzling on pointer dereference, as described here, is called **software swizzling**.

Buffer management is more complicated if pointer swizzling is used, since the physical location of an object must not change once that object is brought into the buffer. One way to ensure that it will not change is to pin pages containing swizzled objects in the buffer pool, so that they are never replaced until the program that performed the swizzling has finished execution. See the bibliographical notes for more complex buffer-management schemes, based on virtual-memory mapping techniques, that make it unnecessary to pin the buffer pages.

11.9.4 Hardware Swizzling

Having two types of pointers, persistent and transient (in-memory), is inconvenient. Programmers have to remember the type of the pointers, and may have to write code twice—once for the persistent pointers and once for the in-memory pointers. It would be simpler if both persistent and in-memory pointers were of the same type.

A simple way to merge persistent and in-memory pointer types is just to extend the length of in-memory pointers to the same size as persistent pointers, and to use 1 bit of the identifier to distinguish between persistent and in-memory pointers. However, the storage cost of longer persistent pointers will have to be borne by in-memory pointers as well; understandably, this scheme is unpopular.

We shall describe a technique called **hardware swizzling**, which uses the virtual-memory-management hardware present in most current computer systems to address this problem. When data in a virtual memory page are accessed, and the operating system detects that the page does not have real storage allocated for it, or has been access protected, then a **segmentation violation** is said to occur.[3] Many operating systems provide a mechanism to specify a function to be called when a segmentation violation occurs, and a mechanism to allocate storage for a page in virtual address space, and to set that page's access permissions. In most Unix systems, the `mmap` system call provides this latter functionality. Hardware swizzling makes clever use of the above mechanisms.

3. The term **page fault** is sometimes used instead of *segmentation violation*, although access protection violations are generally not considered to be page faults.

Hardware swizzling has two major advantages over software swizzling:

1. It is able to store persistent pointers in objects in the same amount of space as in-memory pointers require (along with extra storage external to the object).
2. It transparently converts between persistent pointers and in-memory pointers in a clever and efficient way. Software written to deal with in-memory pointers can thereby deal with persistent pointers as well, without any changes.

11.9.4.1 Pointer Representation

Hardware swizzling uses the following representation of persistent pointers contained in objects that are on disk. A persistent pointer is conceptually split into two parts: a page identifier in the database, and an offset within the page.[4] The page identifier in a persistent pointer is actually a small indirect pointer, which we call the short page identifier. Each page (or other unit of storage) has a translation table that provides a mapping from the short page identifiers to full database page identifiers. The system has to look up the short page identifier in a persistent pointer in the translation table to find the full page identifier.

The translation table, in the worst case, will be only as big as the maximum number of pointers that can be contained in objects in a page; with a page size of 4096, and a pointer size of 4 bytes, the maximum number of pointers is 1024. In practice, the translation table is likely to contain much less than the maximum number of elements (1024 in our example) and will not consume excessive space. The short page identifier needs to have only enough bits to identify a row in the table; with a maximum table size of 1024, only 10 bits are required. Hence, a small number of bits is enough to store the short page identifier. Thus, the translation table permits an entire persistent pointer to fit into the same space as an in-memory pointer. Even though only a few bits are needed for the short page identifier, all the bits of an in-memory pointer, other than the page-offset bits, are used as the short page identifier. This architecture facilitates swizzling, as we shall see.

The persistent-pointer representation scheme appears in Figure 11.22, where there are three objects in the page, each containing a persistent pointer. The translation table gives the mapping between short page identifiers and the full database page identifiers for each of the short page identifiers in these persistent pointers. The database page identifiers are shown in the format *volume.page.offset*.

Each page maintains extra information so that all persistent pointers in the page can be found. The system updates the information when an object is created or deleted in the page. The need to locate all the persistent pointers in a page will become clear later.

4. The term **page** is generally used to refer to a real-memory or virtual-memory page, and the term **block** is used to refer to disk blocks in the database. In hardware swizzling, these have to be of the same size, and database blocks are fetched into virtual memory pages. We shall use the terms *page* and *block* interchangeably in this section.

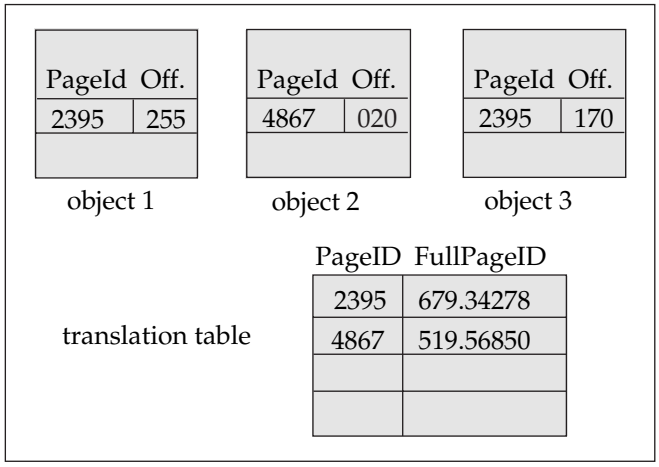


Figure 11.22 Page image before swizzling.

11.9.4.2 Swizzling Pointers on a Page

Initially no page of the database has been allocated a page in virtual memory. Virtual-memory pages may be allocated to database pages even before they are actually loaded, as we will see shortly. Database pages get loaded into virtual-memory when the database system needs to access data on the page. Before a database page is loaded, the system allocates a virtual-memory page to the database page if one has not already been allocated. The system then loads the database page into the virtual-memory page it has allocated to it.

When the system loads a database page P into virtual memory, it does pointer swizzling on the page: It locates all persistent pointers contained in objects in page P, using the extra information stored in the page. It takes the following actions for each persistent pointer in the page. (Let the value of the persistent pointer be $\langle p_i, o_i \rangle$, where p_i is the short page identifier and o_i is the offset within the page. Let P_i be the full page identifier of p_i, found in the translation table in page P.)

1. If page P_i does not already have a virtual-memory page allocated to it, the system now allocates a free page in virtual memory to it. The page P_i will reside at this virtual-memory location if and when it is brought in. At this point, the page in virtual address space does not have any storage allocated for it, either in memory or on disk; it is merely a range of addresses reserved for the database page. The system allocates actual space when it actually loads the database page P_i into virtual memory.

2. Let the virtual-memory page allocated (either earlier or in the preceding step) for P_i be v_i. The system updates the persistent pointer being considered, whose value is $\langle p_i, o_i \rangle$, by replacing p_i with v_i.

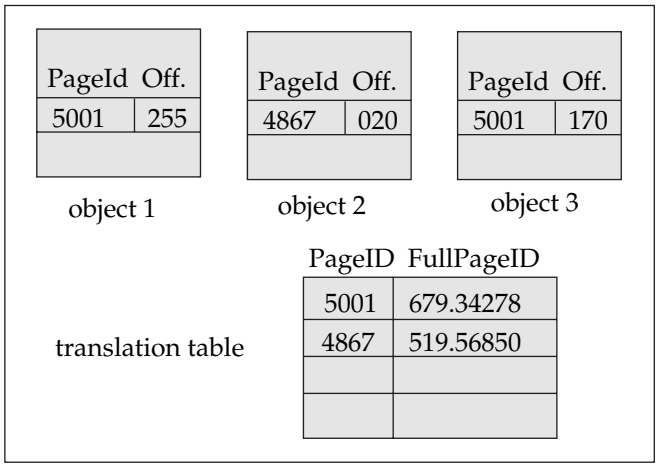


Figure 11.23 Page image after swizzling.

Figure 11.23 shows the state of the page from Figure 11.22 after the system has brought that page into memory and swizzled the pointers in it. Here, we assume that the page whose database page identifier is 679.34278 has been mapped to page 5001 in memory, whereas the page whose identifier is 519.56850 has been mapped to page 4867 (which is the same as the short page identifier). All the pointers in objects have been updated to reflect the new mapping, and can now be used as in-memory pointers.

At the end of the translation phase for a page, the objects in the page satisfy an important property: All persistent pointers contained in objects in the page have been converted to in-memory pointers. Thus, objects in in-memory pages contain only in-memory pointers. Routines that use these objects do not even need to know about the existence of persistent pointers! For example, existing libraries written for in-memory objects can be used unchanged for persistent objects. That is indeed an important advantage!

11.9.4.3 Pointer Dereference

Consider the first time that an in-memory pointer to a virtual-memory page v_i is dereferenced, when storage has not yet been allocated for the page. As we described, a segmentation violation will occur, and will result in a function call on the database system. The database system takes the following actions:

1. It first determines what database page was allocated to virtual-memory page v_i; let the full page identifier of the database page be P_i. (If no database page has been allocated to v_i, the pointer is incorrect, and the system flags an error.)
2. It allocates storage space for page v_i, and loads the database page P_i into virtual-memory page v_i.

3. It carries out pointer swizzling out on page P_i, as described earlier in "Swizzling Pointer on a Page".

4. After swizzling all persistent pointers in P, the system allows the pointer dereference that resulted in the segmentation violation to continue. The pointer dereference will find the object for which it was looking loaded in memory.

If any swizzled pointer that points to an object in page v_i is dereferenced later, the dereference proceeds just like any other virtual-memory access, with no extra overheads. In contrast, if swizzling is not used, there is considerable overhead in locating the buffer page containing the object and then accessing it. This overhead has to be incurred on *every* access to objects in the page, whereas when swizzling is performed, the overhead is incurred only on the *first* access to an object in the page. Later accesses operate at regular virtual-memory access speeds. Hardware swizzling thus gives excellent performance benefits to applications that repeatedly dereference pointers.

11.9.4.4 Optimizations

Software swizzling performs a deswizzling operation when a page in memory has to be written back to the database, to convert in-memory pointers back to persistent pointers. Hardware swizzling can avoid this step—when the system does pointer swizzling for the page, it simply updates the translation table for the page, so that the page-identifier part of the swizzled in-memory pointers can be used to look up the table. For example, as shown in Figure 11.23, database page 679.34278 (with short identifier 2395 in the page shown) is mapped to virtual-memory page 5001. At this point, not only is the pointer in object 1 updated from 2395255 to 5001255, but also the short identifier in the table is updated to 5001. Thus, the short identifier 5001 in object 1 and in the table match each other again. Therefore, the page can be written back to disk without any deswizzling.

Several optimizations can be carried out on the basic scheme described here. When the system swizzles page P, for each page P' referred to by any persistent pointer in P, it attempts to allocate P' to the virtual address location indicated by the short page identifier of P' on page P. If the system can allocate the page in this attempt, pointers to it do not need to be updated. In our swizzling example, page 519.56850 with short page identifier 4867 was mapped to virtual-memory page 4867, which is the same as its short page identifier. We can see that the pointer in object 2 to this page did not need to be changed during swizzling. If every page can be allocated to its appropriate location in virtual address space, none of the pointers need to be translated, and the cost of swizzling is reduced significantly.

Hardware swizzling works even if the database is bigger than virtual memory, but only as long as all the pages that a particular process accesses fit into the virtual memory of the process. If they do not, a page that has been brought into virtual memory will have to be replaced, and that replacement is hard to do, since there may be in-memory pointers to objects in that page.

Hardware swizzling can also be used at the level of sets of pages (often called segments), instead of for a single page. For set-level swizzling, the system uses a single translation table for all pages in the segment. It loads pages in the segment and swizzles them as and when they are required; they need not be loaded all together.

11.9.5 Disk Versus Memory Structure of Objects

The format in which objects are stored in memory may be different from the format in which they are stored on disk in the database. One reason may be the use of software swizzling, where the structures of persistent and in-memory pointers are different. Another reason may be that we want to have the database accessible from different machines, possibly based on different architectures, and from different languages, and from programs compiled under different compilers, all of which result in differences in the in-memory representation.

Consider, for example, a data-structure definition in a programming language such as C++. The physical structure (such as sizes and representation of integers) in the object depends on the machine on which the program is run.[5] Further, the physical structure may also depend on which compiler is used—in a language as complex as C++, different choices for translation from the high-level description to the physical structure are possible, and each compiler can make its own choice.

The solution to this problem is to make the physical representation of objects in the database independent of the machine and of the compiler. The system can convert the object from the disk representation to the form that is required on the specific machine, language, and compiler, when that object is brought into memory. It can do this conversion transparently at the same time that it swizzles pointers in the object, so the programmer does not need to worry about the conversion.

The first step in implementing such a scheme is to define a common language for describing the structure of objects—that is, a data-definition language. One such language is the Object Definition Language (ODL) developed by the Object Database Management Group (ODMG). ODL has mappings defined to the Java, C++, and Smalltalk languages, so potentially we may manipulate objects in an ODMG-compliant database using any of these languages.

The definition of the structure of each class in the database is stored (logically) in the databases. The code to translate an object in the database to the representation that is manipulated with the programming language (and vice versa) depends on the machine as well as on the compiler for the language. We can generate this code automatically, using the stored definition of the class of the object.

An unexpected source of differences between the disk and in-memory representations of data is the hidden-pointers in objects. **Hidden pointers** are transient pointers

5. For instance, the Motorola 680x0 architectures, the IBM 360 architecture, and the Intel 80386/80486/Pentium/Pentium-II/Pentium-III architectures all have 4-byte integers. However, they differ in how the bits of an integer are laid out within a word. In earlier-generation personal computers, integers were 2 bytes long; in newer workstation architectures, such as the Compaq Alpha, Intel Itanium, and Sun UltraSparc architectures, integers can be 8 bytes long.

that compilers generate and store in objects. These pointers point (indirectly) to tables used to implement certain methods of the object. The tables are typically compiled into executable object code, and their exact location depends on the executable object code; hence, they may be different for different processes. Therefore, when a process accesses an object, the hidden pointers must be fixed to point to the correct location. The hidden pointers can be initialized at the same time that data-representation conversions are carried out.

11.9.6 Large Objects

Objects may also be extremely large; for instance, multimedia objects may occupy several megabytes of space. Exceptionally large data items, such as video sequences, may run into gigabytes, although they are usually split into multiple objects, each on the order of a few megabytes or less. Large objects containing binary data are called binary large objects (blobs), while large objects containing character data, are called character large objects (clobs), as we saw in Section 9.2.1.

Most relational databases restrict the size of a record to be no larger than the size of a page, to simplify buffer management and free-space management. Large objects and long fields are often stored in a special file (or collection of files) reserved for long-field storage.

Allocation of buffer pages presents a problem with managing large objects. Large objects may need to be stored in a contiguous sequence of bytes when they are brought into memory; in that case, if an object is bigger than a page, contiguous pages of the buffer pool must be allocated to store it, which makes buffer management more difficult.

We often modify large objects by updating part of the object, or by inserting or deleting parts of the object, rather than by writing the entire object. If inserts and deletes need to be supported, we can handle large objects by using B-tree structures (which we study in Chapter 12). B-tree structures permit us to read the entire object, as well as to insert and delete parts of the object.

For practical reasons, we may manipulate large objects by using application programs, instead of doing so within the database:

- **Text data**. Text is usually treated as a byte string manipulated by editors and formatters.
- **Image/Graphical data**. Graphical data may be represented as a bitmap or as a set of lines, boxes, and other geometric objects. Although some graphical data often are managed within the database system itself, special application software is used for many cases, such as integrated circuit design.
- **Audio and video data**. Audio and video data are typically a digitized, compressed representation created and displayed by separate application software. Data are usually modified with special-purpose editing software, outside the database system.

The most widely used method for updating such data is the **checkout/checkin** method. A user or an application would **check out** a copy of a long-field object, operate on this copy with special-purpose application programs, and then **check in** the modified copy. *Checkout* and a *checkin* correspond roughly to read and write. In some systems, a checkin may create a new version of the object without deleting the old version.

11.10 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 devise.
- 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 (RAIDs). 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. There are different techniques for implementing variable-length records, including the slotted-page method, the pointer method, and the reserved-space method.
- 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.
- 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 avail-

able for storage of copies of disk blocks. The subsystem responsible for the allocation of buffer space is called the *buffer manager*.

- Storage systems for object-oriented databases are somewhat different from storage systems for relational databases: They must deal with large objects, for example, and must support persistent pointers. There are schemes to detect dangling persistent pointers.
- Software- and hardware-based swizzling schemes permit efficient dereferencing of persistent pointers. The hardware-based schemes use the virtual-memory-management support implemented in hardware, and made accessible to user programs by many current-generation operating systems.

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 (NV-RAM)
  - Log disk
  - Log-based file system
- 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
- Buffer
  - Buffer manager
  - Pinned blocks
  - Forced output of blocks
- Buffer-replacement policies
  - Least recently used (LRU)
  - Toss-immediate
  - Most recently used (MRU)
- File
- File organization
  - File header
  - Free list
- Variable-length records
  - Byte-string representation
  - Slotted-page structure
  - Reserved space
  - List representation
- Heap file organization
- Sequential file organization
- Hashing file organization
- Clustering file organization
- Search key
- Data dictionary
- System catalog
- Storage structures for OODBs
- Object identifier (OID)
  - Logical OID
  - Physical OID
  - Unique identifier
  - Dangling pointer
  - Forwarding address
- Pointer swizzling
  - Dereferencing
  - Deswizzling
  - Software swizzling
  - Hardware swizzling
  - Segmentation violation
  - Page fault
- Hidden pointers
- Large objects

Exercises

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

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

11.3 Consider the following data and parity-block arrangement on four disks:

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}
⋮	⋮	⋮	⋮

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?

11.4 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)

11.5 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. With which of the RAID levels is the amount of interference between the rebuild and ongoing disk accesses least? Explain your answer.

11.6 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.

11.7 Consider the deletion of record 5 from the file of Figure 11.8. 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.

11.8 Show the structure of the file of Figure 11.9 after each of the following steps:

a. Insert (Brighton, A-323, 1600).
b. Delete record 2.
c. Insert (Brighton, A-626, 2000).

11.9 Give an example of a database application in which the reserved-space method of representing variable-length records is preferable to the pointer method. Explain your answer.

11.10 Give an example of a database application in which the pointer method of representing variable-length records is preferable to the reserved-space method. Explain your answer.

11.11 Show the structure of the file of Figure 11.12 after each of the following steps:

a. Insert (Mianus, A-101, 2800).
b. Insert (Brighton, A-323, 1600).
c. Delete (Perryridge, A-102, 400).

11.12 What happens if you attempt to insert the record

(Perryridge, A-929, 3000)

into the file of Figure 11.12?

11.13 Show the structure of the file of Figure 11.13 after each of the following steps:

a. Insert (Mianus, A-101, 2800).

b. Insert (Brighton, A-323, 1600).

c. Delete (Perryridge, A-102, 400).

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

11.15 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.

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

11.17 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.

11.18 Consider a relational database with two relations:

course (*course-name*, *room*, *instructor*)
enrollment (*course-name*, *student-name*, *grade*)

Define instances of these relations for three courses, each of which enrolls five students. Give a file structure of these relations that uses clustering.

11.19 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 when searching for free space and when updating free space information.

11.20 Give a normalized version of the *Index-metadata* relation, and explain why using the normalized version would result in worse performance.

11.21 Explain why a physical OID must contain more information than a pointer to a physical storage location.

11.22 If physical OIDs are used, an object can be relocated by keeping a forwarding pointer to its new location. In case an object gets forwarded multiple times, what would be the effect on retrieval speed? Suggest a technique to avoid multiple accesses in such a case.

11.23 Define the term *dangling pointer*. Describe how the unique-id scheme helps in detecting dangling pointers in an object-oriented database.

11.24 Consider the example on page 435, which shows that there is no need for deswizzling if hardware swizzling is used. Explain why, in that example, it is safe to change the short identifier of page 679.34278 from 2395 to 5001. Can some other page already have short identifier 5001? If it could, how can you handle that situation?

Bibliographical Notes

Patterson and Hennessy [1995] discusses the hardware aspects of translation look-aside buffers, caches and memory-management units. Rosch and Wethington [1999] presents an excellent overview of computer hardware, including extensive coverage of all types of storage technology such as floppy disks, magnetic disks, optical disks, tapes, and storage interfaces. Ruemmler and Wilkes [1994] presents a survey of magnetic-disk technology. Flash memory is discussed by Dippert and Levy [1993].

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

Alternative disk organizations that provide a high degree of fault tolerance include those described by Gray et al. [1990] and Bitton and Gray [1988]. Disk striping is described by Salem and Garcia-Molina [1986]. Discussions of redundant arrays of inexpensive disks (RAID) are presented by Patterson et al. [1988] and Chen and Patterson [1990]. Chen et al. [1994] presents an excellent survey of RAID principles and implementation. Reed–Solomon codes are covered in Pless [1989]. The log-based file system, which makes disk access sequential, is described in Rosenblum and Ousterhout [1991].

In systems that support mobile computing, data may be broadcast repeatedly. The broadcast medium can be viewed as a level of the storage hierarchy—as a broadcast disk with high latency. These issues are discussed in Acharya et al. [1995]. Caching and buffer management for mobile computing is discussed in Barbará and Imielinski [1994]. Further discussion of storage issues in mobile computing appears in Douglis et al. [1994].

Basic data structures are discussed in Cormen et al. [1990]. There are several papers describing the storage structure of specific database systems. Astrahan et al. [1976] discusses System R. Chamberlin et al. [1981] reviews System R in retrospect. The *Oracle 8 Concepts Manual* (Oracle [1997]) describes the storage organization of the Oracle 8 database system. The structure of the Wisconsin Storage System (WiSS) is described in Chou et al. [1985]. A software tool for the physical design of relational databases is described by Finkelstein et al. [1988].

Buffer management is discussed in most operating system texts, including in Silberschatz and Galvin [1998]. Stonebraker [1981] discusses the relationship between database-system buffer managers and operating-system buffer managers. Chou and Dewitt [1985] presents algorithms for buffer management in database systems, and describes a performance evaluation. Bridge et al. [1997] describes techniques used in the buffer manager of the Oracle database system.

Descriptions and performance comparisons of different swizzling techniques are given in Wilson [1990], Moss [1990], and White and DeWitt [1992]. White and DeWitt [1994] describes the virtual-memory-mapped buffer-management scheme used in the ObjectStore OODB system and in the QuickStore storage manager. Using this scheme, we can map disk pages to a fixed virtual-memory address, even if they are not pinned in the buffer. The Exodus object storage manager is described in Carey et al. [1986]. Biliris and Orenstein [1994] provides a survey of storage systems for object-oriented databases. Jagadish et al. [1994] describes a storage manager for main-memory databases.

CHAPTER 12

Indexing and Hashing

Many queries reference only a small proportion of the records in a file. For example, a query like "Find all accounts at the Perryridge branch" or "Find the balance of account number A-101" references only a fraction of the account records. It is inefficient for the system to read every record and to check the *branch-name* field for the name "Perryridge," or the *account-number* field for the value A-101. 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.

12.1 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 we are looking for. The words in the index are in sorted order, making it easy to find the word we are looking for. Moreover, the index is much smaller than the book, further reducing the effort needed to find the words we are looking for.

Card catalogs in libraries worked in a similar manner (although they are rarely used any longer). To find a book by a particular author, we would search in the author catalog, and a card in the catalog tells us where to find the book. To assist us in searching the catalog, the library would keep the cards in alphabetic order by authors, with one card for each author of each book.

Database system indices play the same role as book indices or card catalogs in libraries. For example, to retrieve an *account* record given the account number, 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 *account* record.

Keeping a sorted list of account numbers would not work well on very large databases with millions of accounts, since the index would itself be very big; further,

even though keeping the index sorted reduces the search time, finding an account 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, libraries maintained several card catalogs: for author, for subject, and for 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.

12.2 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 deci-

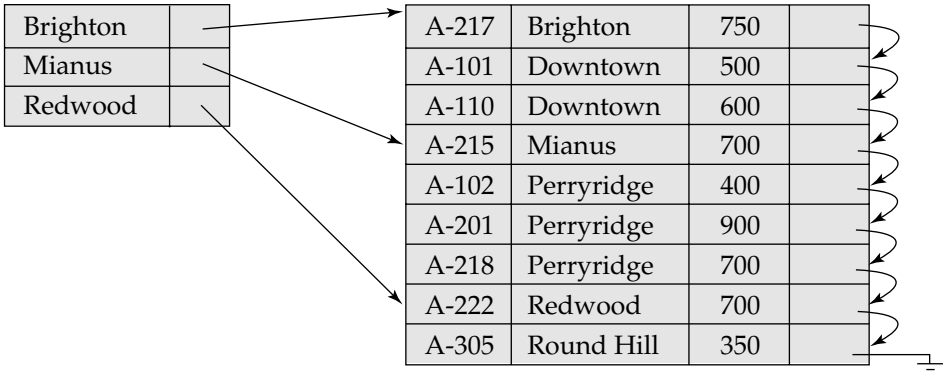


Figure 12.1 Sequential file for *account* records.

mal number. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **primary index** is an index whose search key also defines the sequential order of the file. (The term *primary index* is sometimes used to mean an index on a primary key. However, such usage is nonstandard and should be avoided.) Primary indices are also called **clustering indices**. The search key of a primary index is usually 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 **secondary indices**, or **nonclustering** indices.

12.2.1 Primary Index

In this section, we assume that all files are ordered sequentially on some search key. Such files, with a primary 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.

Figure 12.1 shows a sequential file of *account* records taken from our banking example. In the example of Figure 12.1, the records are stored in search-key order, with *branch-name* used as the search key.

12.2.1.1 Dense and Sparse Indices

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

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

- **Dense index**: An index record appears for every search-key value in the file. In a dense primary 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 primary one, records are sorted on the same search key.

Dense index implementations may store a list of pointers to all records with the same search-key value; doing so is not essential for primary indices.

- **Sparse index**: An index record appears for only some of the search-key values. As is true in dense indices, each index record 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 12.2 and 12.3 show dense and sparse indices, respectively, for the *account* file. Suppose that we are looking up records for the Perryridge branch. Using the dense index of Figure 12.2, we follow the pointer directly to the first Perryridge record. We process this record, and follow the pointer in that record to locate the next record in search-key (*branch-name*) order. We continue processing records until we encounter a record for a branch other than Perryridge. If we are using the sparse index (Figure 12.3), we do not find an index entry for "Perryridge." Since the last entry (in alphabetic order) before "Perryridge" is "Mianus," we follow that pointer. We then read the *account* file in sequential order until we find the first Perryridge record, and begin processing at that point.

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 pro-

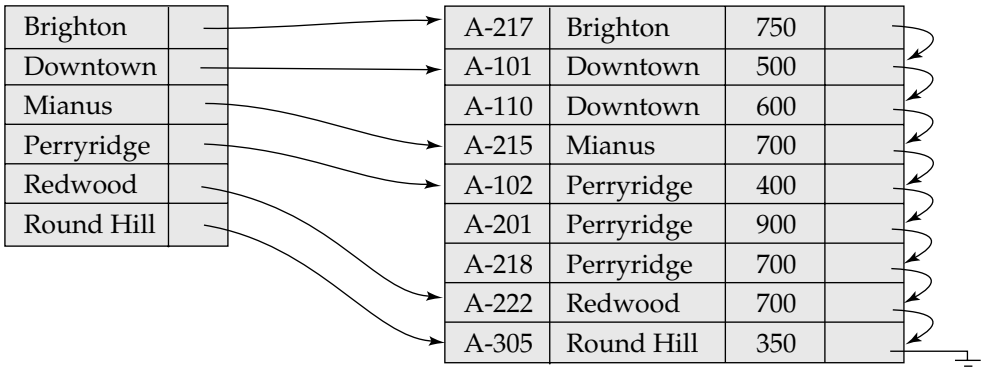


Figure 12.2 Dense index.

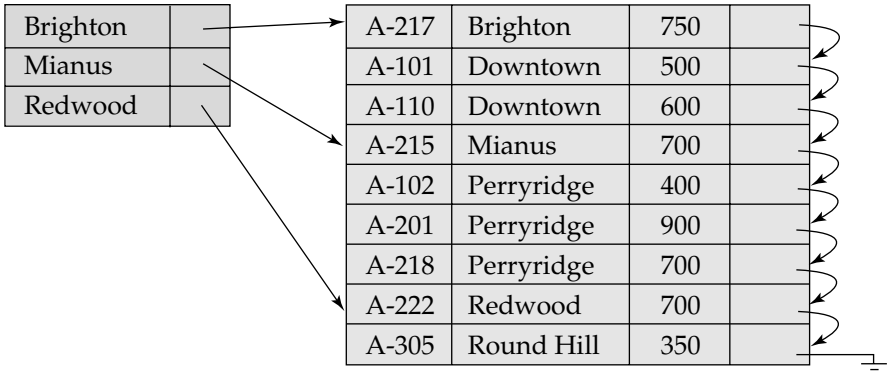


Figure 12.3 Sparse index.

cessing 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 11.7.1), we minimize block accesses while keeping the size of the index (and thus, our space overhead) as small as possible.

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.

12.2.1.2 Multilevel Indices

Even if we use a sparse index, the index itself may become too large for efficient processing. It is not unreasonable, in practice, to have a file with 100,000 records, with 10 records stored in each block. If we have one index record per block, the index has 10,000 records. Index records are smaller than data records, so let us assume that 100 index records fit on a block. Thus, our index occupies 100 blocks. Such large indices are stored as sequential files on disk.

If an index is sufficiently small to be kept in main memory, the search time to find an entry is low. However, if the index is so large that it must be kept on disk, a search for an entry 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 occupies 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 our 100-block index, binary search requires seven block reads. On a disk system where a block read takes 30 milliseconds, the search will take 210 milliseconds, which is long. Note that, if overflow blocks have been used, binary search will not be 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.

To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse index on the primary index, as in Figure 12.4.

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.

Using the two levels of indexing, we have read only one index block, rather than the seven we read with binary search, if we assume that the outer index is already in main memory. If our file is extremely large, even the outer index may grow too large to fit in main memory. 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. Each level of index could correspond to a unit of physical storage. Thus, we may have indices at the track, cylinder, and disk levels.

A typical dictionary is an example of a multilevel index in the nondatabase world. The header of each page lists the first word alphabetically on that page. Such a book

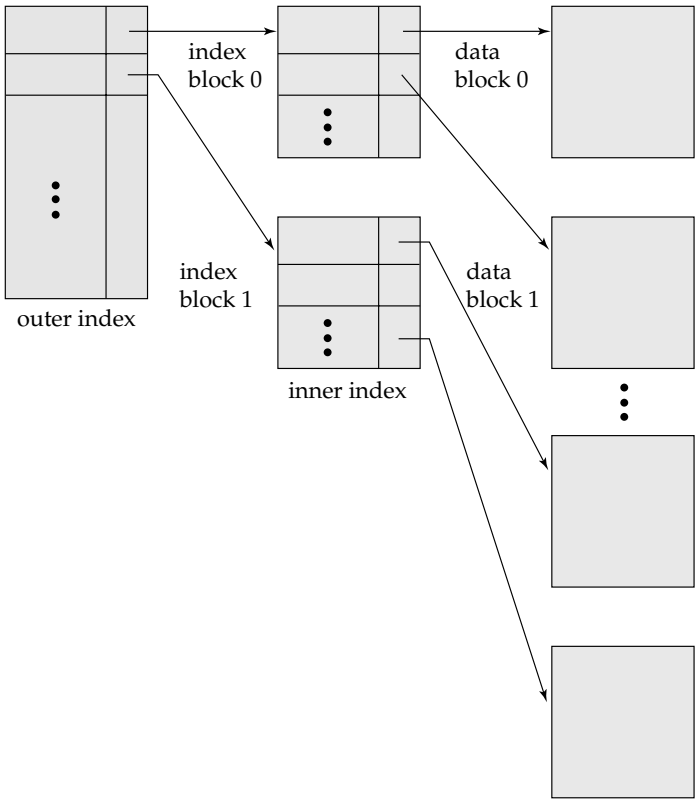


Figure 12.4 Two-level sparse index.

index is a multilevel index: The words at the top of each page of the book index form a sparse index on the contents of the dictionary pages.

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 12.3.

12.2.1.3 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. 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. Again, 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 record with the search-key value in the index at the appropriate position.
    2. Otherwise the following actions are taken:
       a. If the index record stores pointers to all records with the same search-key value, the system adds a pointer to the new record to the index record.
       b. Otherwise, the index record 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 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 record from the index.
    2. Otherwise the following actions are taken:
       a. If the index record stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index record.

b. Otherwise, the index record 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 record to point to the next record.

- Sparse indices:
  1. If the index does not contain an index record 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 record for the search-key value points to the record being deleted, the system updates the index record 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.

12.2.2 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 primary 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 primary 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 primary indices. If the search key of a primary 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 primary index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

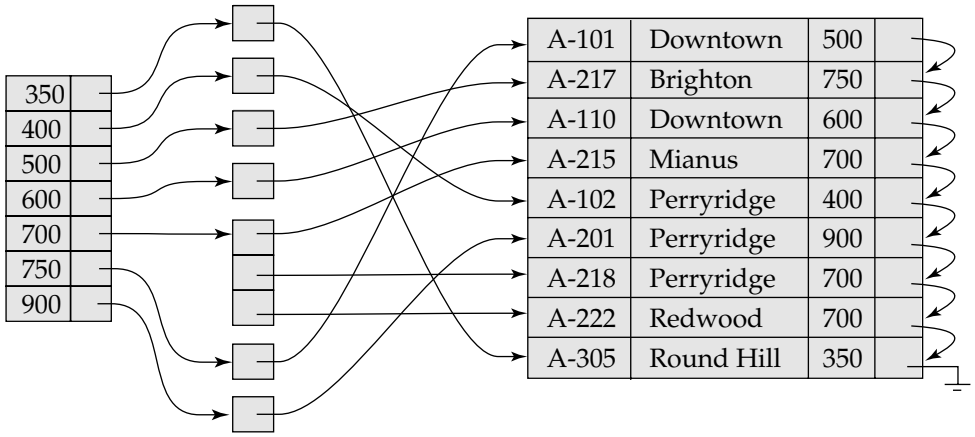


Figure 12.5 Secondary index on *account* file, on noncandidate key *balance*.

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 12.5 shows the structure of a secondary index that uses an extra level of indirection on the *account* file, on the search key *balance*.

A sequential scan in primary 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 both by the search key of the primary index, and the search key of a secondary index. 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 primary 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.

12.3 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.

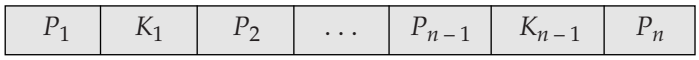


Figure 12.6 Typical node of a B$^+$-tree.

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.

12.3.1 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 12.6 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 either a file record with search-key value K_i or to a bucket of pointers, each of which points to a file record with search-key value K_i. The bucket structure is used only if the search key does not form a primary key, and if the file is not sorted in the search-key value order. Pointer P_n has a special purpose that we shall discuss shortly.

Figure 12.7 shows one leaf node of a B$^+$-tree for the *account* file, in which we have chosen n to be 3, and the search key is *branch-name*. Note that, since the account file is ordered by *branch-name*, the pointers in the leaf node point directly to the file.

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. The ranges of values in each leaf do not overlap. Thus, if L_i and L_j are leaf nodes and $i < j$, then every search-key value in L_i is less than every search-key value in L_j. If the B$^+$-tree index is to be a dense index, 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 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*

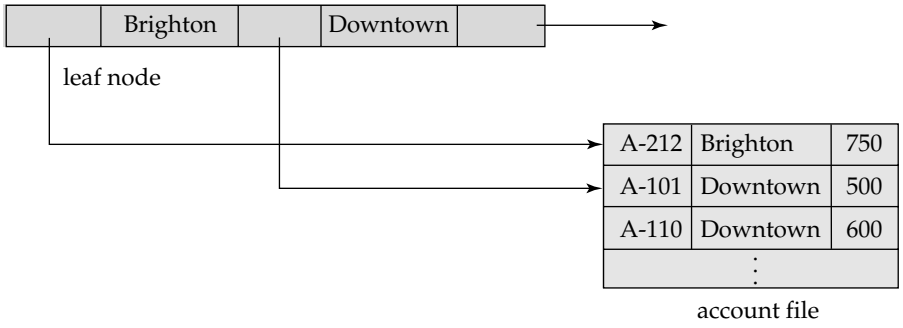


Figure 12.7 A leaf node for *account* B+-tree index ($n = 3$).

hold at least $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fanout* of the node.

Let us consider a node containing m pointers. 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 12.8 shows a complete B+-tree for the *account* file ($n = 3$). For simplicity, we have omitted both the pointers to the file itself and the null pointers. As an example of a B+-tree for which the root must have less than $\lceil n/2 \rceil$ values, Figure 12.9 shows a B+-tree for the *account* file with $n = 5$.

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.

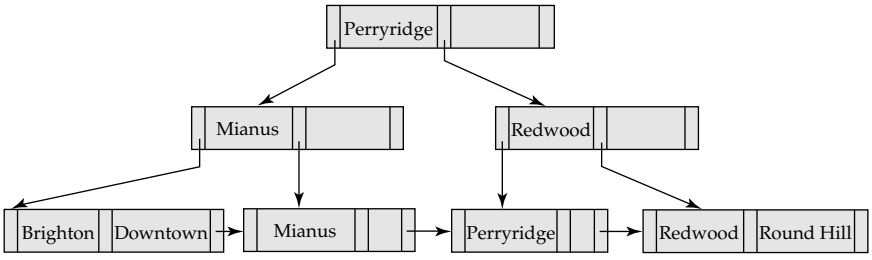


Figure 12.8 B+-tree for *account* file ($n = 3$).

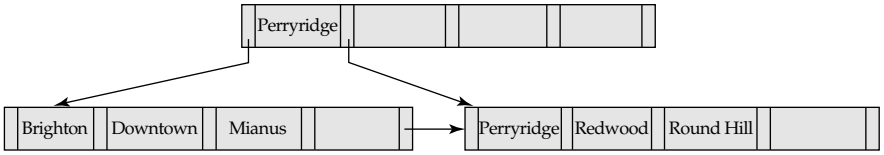


Figure 12.9 B$^+$-tree for *account* file with $n = 5$.

12.3.2 Queries on B$^+$-Trees

Let us consider how we process queries on a B$^+$-tree. Suppose that we wish to find all records with a search-key value of V. Figure 12.10 presents pseudocode for doing so. Intuitively, the procedure works as follows. First, we examine the root node, looking for the smallest search-key value greater than V. Suppose that we find that this search-key value is K_i. We then follow pointer P_i to another node. If we find no such value, then $k \geq K_{m-1}$, where m is the number of pointers in the node. In this case we follow P_m to another node. In the node we reached above, again we look for the smallest search-key value greater than V, and once again follow the corresponding pointer as above. Eventually, we reach a leaf node. At the leaf node, if we find search-key value K_i equals V, then pointer P_i directs us to the desired record or bucket. If the value V is not found in the leaf node, no record with key value V exists.

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

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

```
procedure find(value V)
    set C = root node
    while C is not a leaf node begin
        Let K_i = smallest search-key value, if any, greater than V
        if there is no such value then begin
            Let m = the number of pointers in the node
            set C = node pointed to by P_m
        end
            else set C = the node pointed to by P_i
    end
    if there is a key value K_i in C such that K_i = V
        then pointer P_i directs us to the desired record or bucket
        else no record with key value k exists
end procedure
```

Figure 12.10 Querying a B$^+$-tree.

$\lceil \log_{50}(1{,}000{,}000) \rceil = 4$ 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 $\lceil \log_2(K) \rceil$, where K is the number of search-key values. With $K = 1{,}000{,}000$ 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.

12.3.3 Updates on B$^+$-Trees

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, we find the leaf node in which the search-key value would appear. If the search-key value already appears in the leaf node, we add the new record to the file and, if necessary, add to the bucket a pointer to the record. If the search-key value does not appear, we insert the value in the leaf node, and position it such that the search keys are still in order. We then insert the new record in the file and, if necessary, create a new bucket with the appropriate pointer.
- **Deletion**. Using the same technique as for lookup, we find the record to be deleted, and remove it from the file. We remove the search-key value from the leaf node if there is no bucket associated with that search-key value or if the bucket becomes empty as a result of the deletion.

We now consider an example in which a node must be split. Assume that we wish to insert a record with a *branch-name* value of "Clearview" into the B$^+$-tree of Figure 12.8. Using the algorithm for lookup, we find that "Clearview" should appear in the node containing "Brighton" and "Downtown." There is no room to insert the search-key value "Clearview." Therefore, the node is *split* into two nodes. Figure 12.11 shows the two leaf nodes that result from inserting "Clearview" and splitting the node containing "Brighton" and "Downtown." In general, we take the n search-key

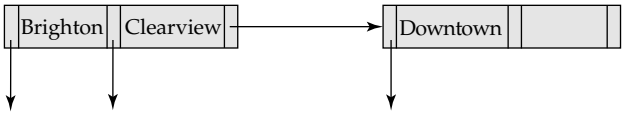


Figure 12.11 Split of leaf node on insertion of "Clearview."

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 new 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 "Downtown" as its smallest search-key value. We need to insert this search-key value into the parent of the leaf node that was split. The B$^+$-tree of Figure 12.12 shows the result of the insertion. The search-key value "Downtown" was inserted into the parent. It was possible to perform this insertion because there was room for an added search-key value. If there were no room, the parent would have had to be split. 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.

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 12.13 outlines the insertion algorithm in pseudocode. In the pseudocode, $L.K_i$ and $L.P_i$ denote the ith value and the ith pointer in node L, respectively. The pseudocode also makes use of the function $parent(L)$ to find the parent of a node L. 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 pseudocode refers to inserting an entry (V, P) into a node. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the leaf node actually stores P before V. For internal nodes, P is stored just after V.

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete "Downtown" from the B$^+$-tree of Figure 12.12. We locate the entry for "Downtown" by using our lookup algorithm. When we delete the entry for "Downtown" from its leaf node, the leaf becomes empty. Since, in our example, $n = 3$ and $0 < \lceil (n-1)/2 \rceil$, this node must be eliminated from the B$^+$-tree. To delete a leaf node,

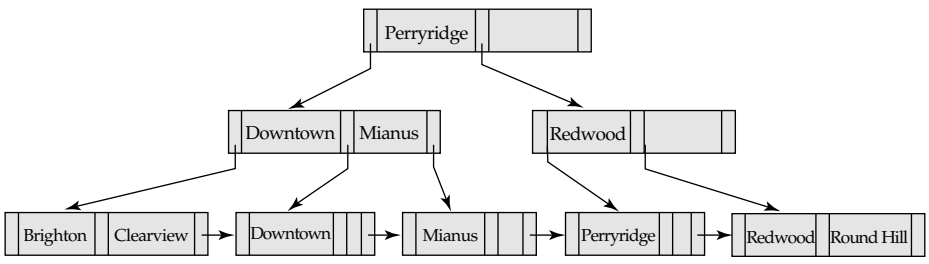


Figure 12.12 Insertion of "Clearview" into the B$^+$-tree of Figure 12.8.

```
procedure insert(value V, pointer P)
    find the leaf node L that should contain value V
    insert_entry(L, V, P)
end procedure

procedure insert_entry(node L, value V, pointer P)
    if (L has space for (V, P))
        then insert (V, P) in L
        else begin /* Split L */
            Create node L'
            Let V' be the value in L.K_1, ..., L.K_{n-1}, V such that exactly
                ⌈n/2⌉ of the values L.K_1, ..., L.K_{n-1}, V are less than V'
            Let m be the lowest value such that L.K_m ≥ V'
            /* Note: V' must be either L.K_m or V */
            if (L is a leaf) then begin
                move L.P_m, L.K_m, ..., L.P_{n-1}, L.K_{n-1} to L'
                if (V < V') then insert (P, V) in L
                else insert (P, V) in L'
            end
            else begin
                if (V = V') /* V is smallest value to go to L' */
                    then add P, L.K_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n to L'
                    else add L.P_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n to L'
                delete L.K_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n from L
                if (V < V') then insert (V, P) in L
                else if (V > V') then insert (V, P) in L'
                /* Case of V = V' handled already */
            end
            if (L is not the root of the tree)
                then insert_entry(parent(L), V', L');
                else begin
                    create a new node R with child nodes L and L' and
                        the single value V'
                    make R the root of the tree
                end
            if (L) is a leaf node then begin /* Fix next child pointers */
                set L'.P_n = L.P_n;
                set L.P_n = L'
            end
        end
end procedure
```

Figure 12.13 Insertion of entry in a B$^+$-tree.

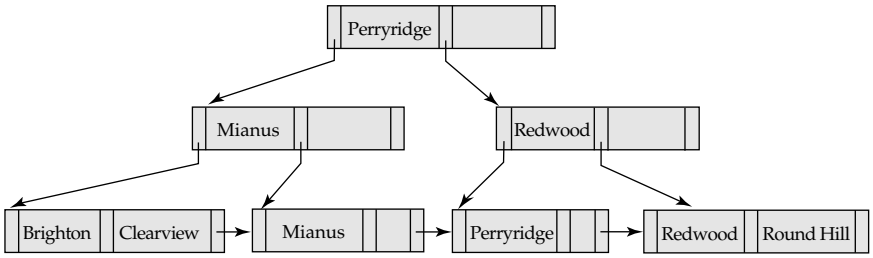


Figure 12.14 Deletion of "Downtown" from the B$^+$-tree of Figure 12.12.

we must delete the pointer to it from its parent. In our example, this deletion leaves the parent node, which formerly contained three pointers, with only two pointers. Since $2 \geq \lceil n/2 \rceil$, the node is still sufficiently large, and the deletion operation is complete. The resulting B$^+$-tree appears in Figure 12.14.

When we make a deletion from a parent of a leaf node, the parent node itself may become too small. That is exactly what happens if we delete "Perryridge" from the B$^+$-tree of Figure 12.14. Deletion of the Perryridge entry causes a leaf node to become empty. When we delete the pointer to this node in the latter's parent, the parent is left with only one pointer. Since $n = 3$, $\lceil n/2 \rceil = 2$, and thus only one pointer is too few. However, since the parent node contains useful information, we cannot simply delete it. Instead, we look at the sibling node (the nonleaf node containing the one search key, Mianus). This sibling node has room to accommodate the information contained in our now-too-small node, so we coalesce these nodes, such that the sibling node now contains the keys "Mianus" and "Redwood." The other node (the node containing only the search key "Redwood") now contains redundant information and can be deleted from its parent (which happens to be the root in our example). Figure 12.15 shows the result. Notice that the root has only one child pointer after the deletion, so it is deleted and its sole child becomes the root. So the depth of the B$^+$-tree has been decreased by 1.

It is not always possible to coalesce nodes. As an illustration, delete "Perryridge" from the B$^+$-tree of Figure 12.12. In this example, the "Downtown" entry is still part of the tree. Once again, the leaf node containing "Perryridge" becomes empty. The parent of the leaf node becomes too small (only one pointer). However, in this example, the sibling node already contains the maximum number of pointers: three. Thus, it cannot accommodate an additional pointer. The solution in this case is to **redistribute** the pointers such that each sibling has two pointers. The result appears in

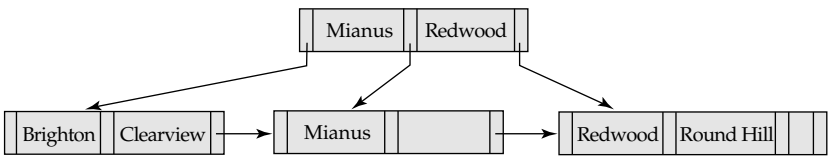


Figure 12.15 Deletion of "Perryridge" from the B$^+$-tree of Figure 12.14.

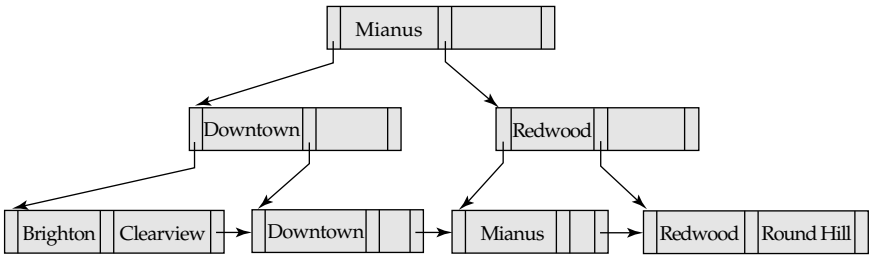


Figure 12.16 Deletion of "Perryridge" from the B+-tree of Figure 12.12.

Figure 12.16. Note that the redistribution of values necessitates a change of a search-key value in the parent of the two siblings.

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 12.17 outlines the pseudocode for deletion from a B+-tree. The procedure swap_variables(L, L') merely swaps the values of the (pointer) variables L and L'; 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 (V, 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 V. For internal nodes, P follows the key value V.

It is worth noting that, as a result of deletion, a key value that is present in an internal node of the B+-tree may not be present at any leaf of the tree.

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 for a worst-case insertion or deletion is proportional to $\log_{\lceil n/2 \rceil}(K)$, where n is the maximum number of pointers in a node, and K is the number of search-key values. In other words, the cost of insertion and deletion 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.

12.3.4 B+-Tree File Organization

As mentioned in Section 12.3, the main drawback of index-sequential file organization is the degradation of performance as the file grows: With growth, an increasing percentage of index records 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

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

Figure 12.17 Deletion of entry from a B^+-tree.

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 12.18 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.

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 internal 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.)

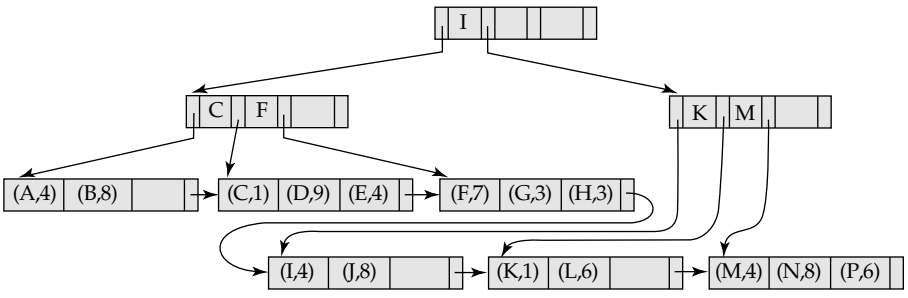


Figure 12.18 B+-tree file organization.

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.

12.4 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 12.12, the search keys "Downtown," "Mianus," "Redwood," and "Perryridge" appear twice. 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. Figure 12.19 shows a B-tree that represents the same search keys as the B$^+$-tree of Figure 12.12. 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 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.

A generalized B-tree leaf node appears in Figure 12.20a; a nonleaf node appears in Figure 12.20b. 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

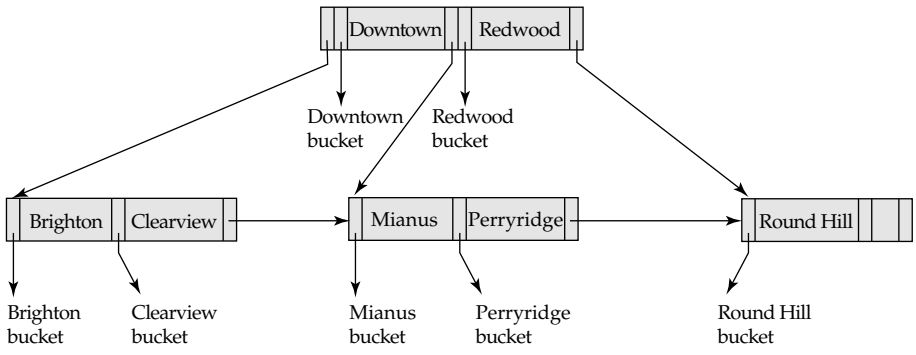


Figure 12.19 B-tree equivalent of B$^+$-tree in Figure 12.12.

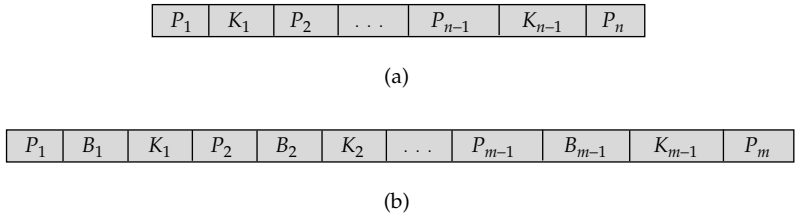


Figure 12.20 Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

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 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, many database system implementers prefer the structural simplicity of a B$^+$-tree. The exercises explore details of the insertion and deletion algorithms for B-trees.

12.5 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.

12.5.1 Hash File Organization

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 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.

12.5.1.1 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 *account* file using the search key *branch-name*. The hash function that we choose must have

the desirable properties not only on the example *account* file that we have been using, but also on an *account* file of realistic size for a large bank with many branches.

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 branch 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 *balance*. Suppose that the minimum balance is 1 and the maximum balance is 100,000, and we use a hash function that divides the values into 10 ranges, 1–10,000, 10,001–20,000 and so on. The distribution of search-key values is uniform (since each bucket has the same number of different *balance* values), but is not random. But records with balances between 1 and 10,000 are far more common than are records with balances between 90,001 and 100,000. 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 12.21 shows the application of such a scheme, with 10 buckets, to the *account* file, under the assumption that the *i*th letter in the alphabet is represented by the integer *i*.

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.

12.5.1.2 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:

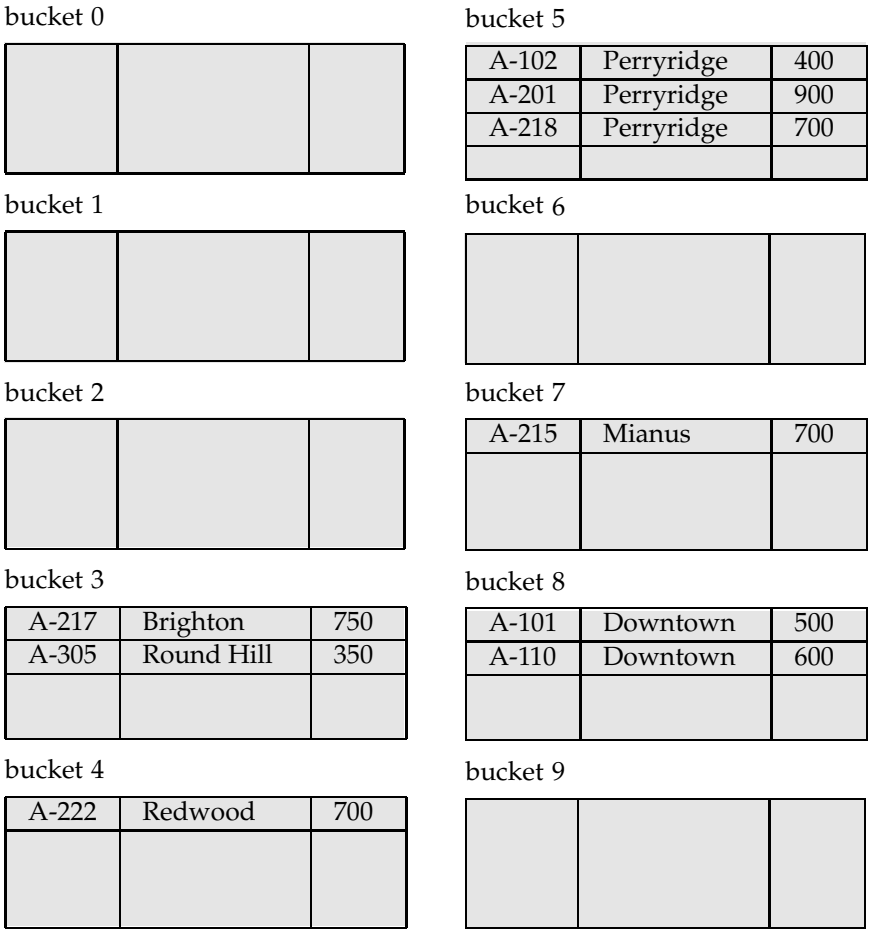


Figure 12.21 Hash organization of *account* file, with *branch-name* as the key.

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 buck-

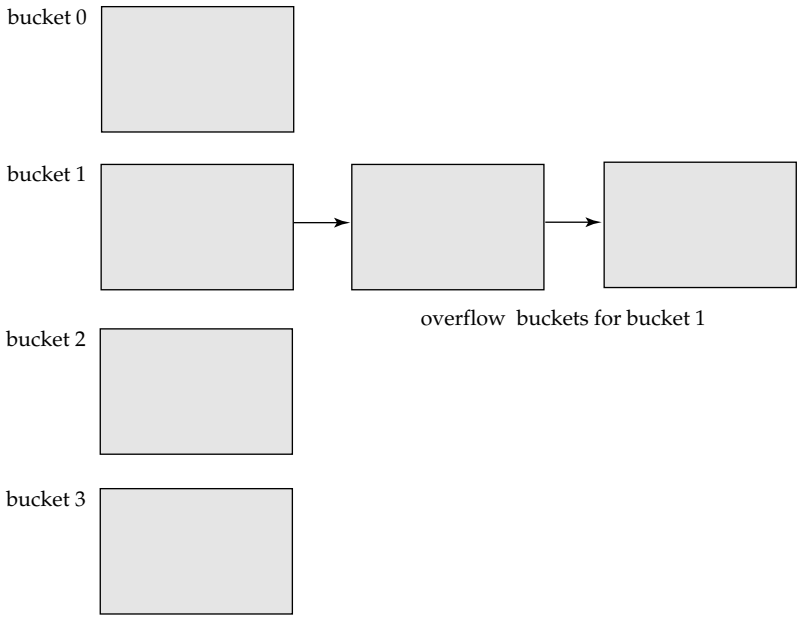


Figure 12.22 Overflow chaining in a hash structure.

ets of a given bucket are chained together in a linked list, as in Figure 12.22. 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 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 12.6, how the number of buckets and the hash function can be changed dynamically.

12.5.2 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 12.23 shows a secondary hash index on the *account* file, for the search key *account-number*. The hash function in the figure computes the sum of the digits of the account number modulo 7. The hash index has seven buckets, each of size 2 (realistic indices would, of course, have much larger

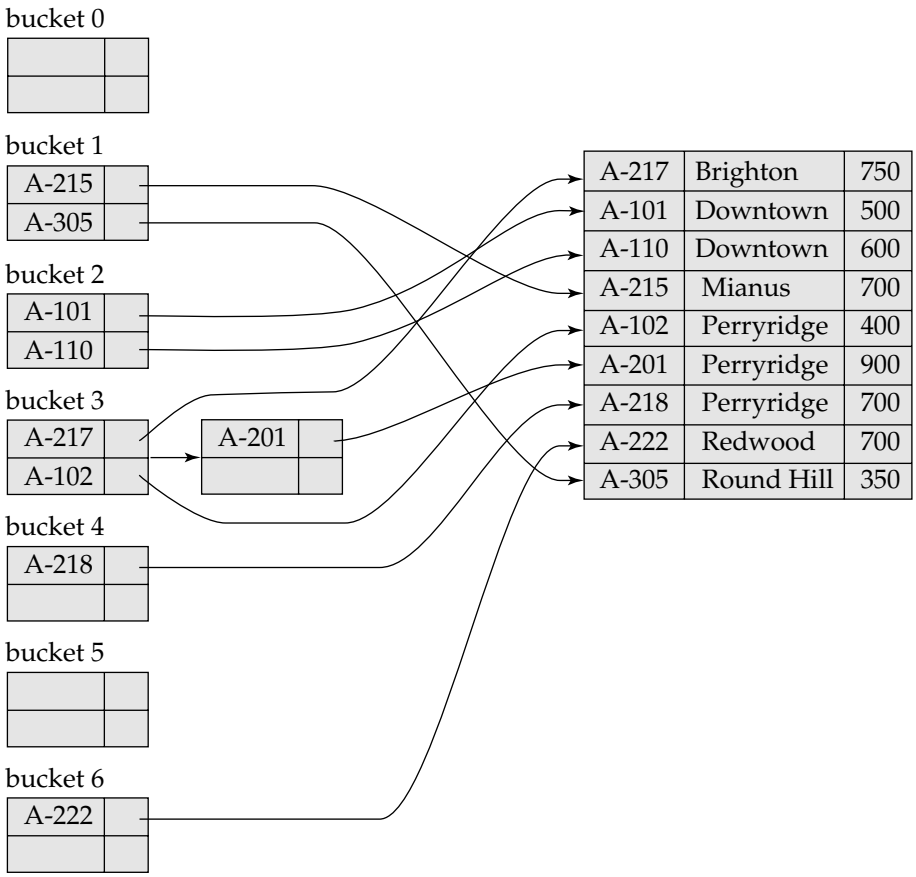


Figure 12.23 Hash index on search key *account-number* of *account* file.

bucket sizes). One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *account-number* is a primary key for *account*, 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 primary 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 primary hash index on it.

12.6 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.

12.6.1 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.

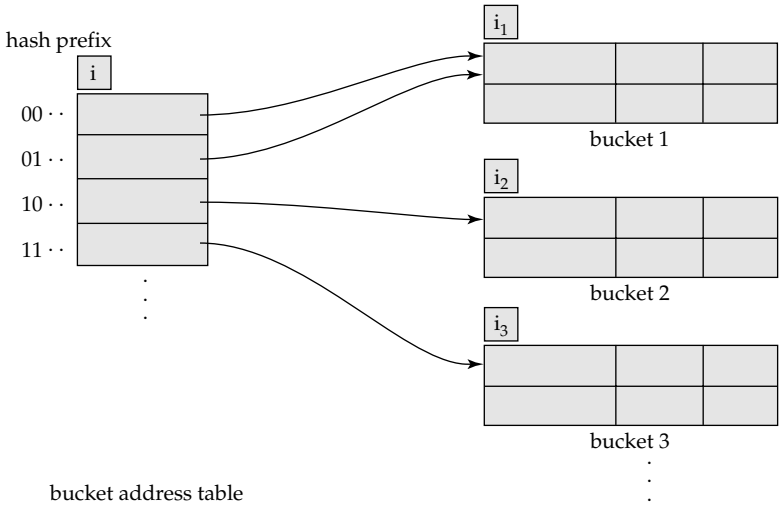


Figure 12.24 General extendable hash structure.

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 12.24 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 12.24 the integer associated with bucket j is shown as i_j. The number of bucket-address-table entries that point to bucket j is

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

12.6.2 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 set 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*.

  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

A-217	Brighton	750
A-101	Downtown	500
A-1 10	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Figure 12.25 Sample *account* file.

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.

Our example *account* file in Figure 12.25 illustrates the operation of insertion. The 32-bit hash values on *branch-name* appear in Figure 12.26. Assume that, initially, the file is empty, as in Figure 12.27. 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.

We insert the record (A-217, Brighton, 750). The bucket address table contains a pointer to the one bucket, and the system inserts the record. Next, we insert the record (A-101, Downtown, 500). The system also places this record in the one bucket of our structure.

When we attempt to insert the next record (Downtown, A-110, 600), 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

branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

Figure 12.26 Hash function for *branch-name*.

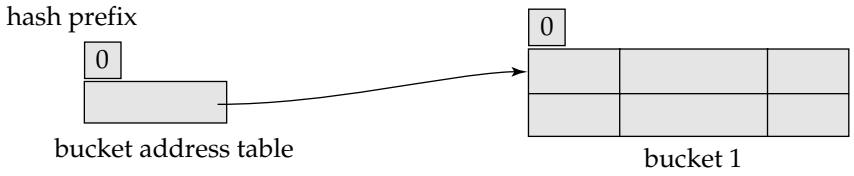


Figure 12.27 Initial extendable hash structure.

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 12.28 shows the state of our structure after the split.

Next, we insert (A-215, Mianus, 700). Since the first bit of h(Mianus) is 1, we must insert this record into the bucket pointed to by the "1" entry in the 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 12.29. Since the bucket of Figure 12.28 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 12.28 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 (A-102, Perryridge, 400), which goes in the same bucket as Mianus. The following insertion, of (A-201, Perryridge, 900), 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. The insertion of the third Perryridge record, (A-218, Perryridge, 700), leads to another overflow. However, 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 12.30.

We continue in this manner until we have inserted all the *account* records of Figure 12.25. The resulting structure appears in Figure 12.31.

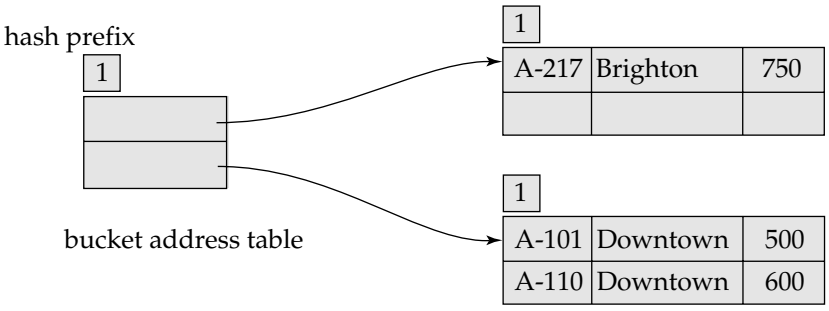


Figure 12.28 Hash structure after three insertions.

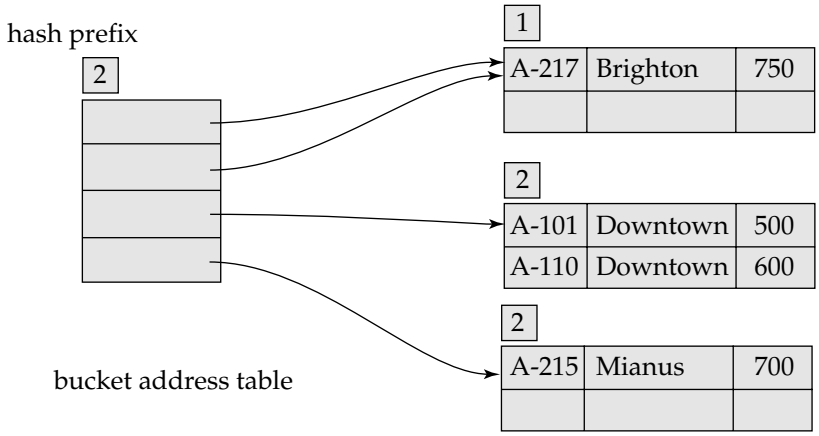


Figure 12.29 Hash structure after four insertions.

12.6.3 Comparison with Other Schemes

We now examine the advantages and disadvantages of extendable hashing, compared with the other schemes that we have discussed. 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 pre-

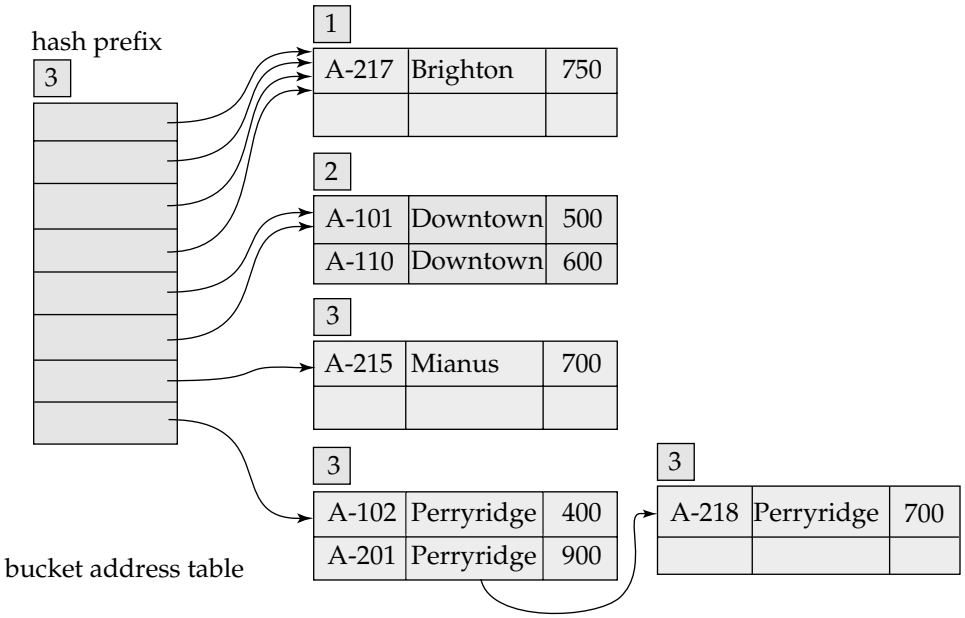


Figure 12.30 Hash structure after seven insertions.

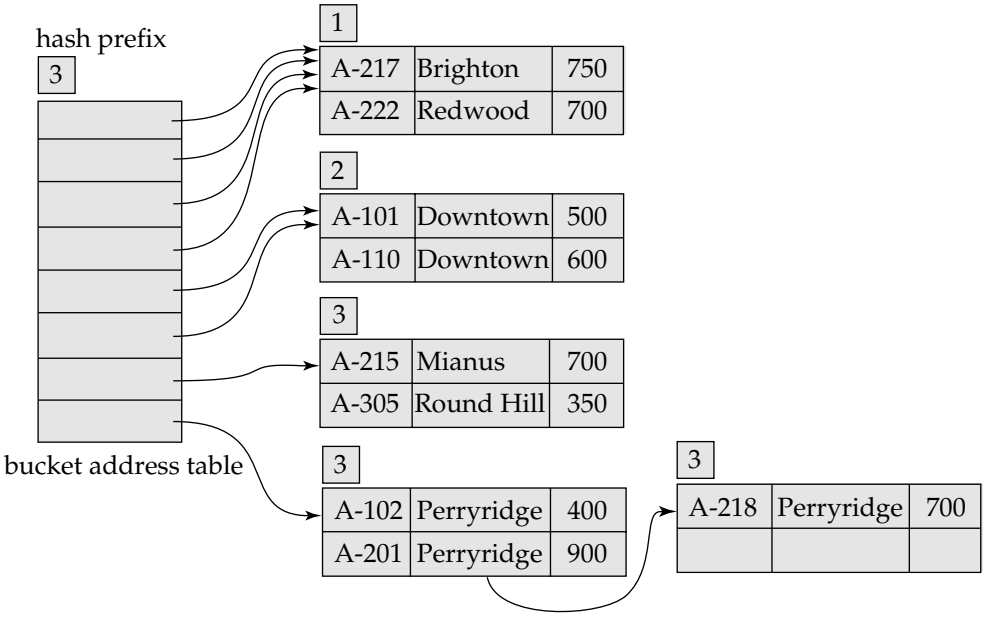


Figure 12.31 Extendable hash structure for the *account* file.

fix 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 12.5 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.

12.7 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 wise 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.

12.8 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. For example, typical implementations enforce a key declaration (Chapter 6) by creating an index with the declared key as the search key of the index.

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:1999 standard. The SQL standards (up to SQL:1999, at least) do not support control of the physical database schema, and have restricted themselves to the logical database schema.

We create an index by 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 name *b-index* on the *branch* relation with *branch-name* as the search key, we write

create index *b-index* **on** *branch* (*branch-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 *b-index* **on** *branch* (*branch-name*)

declares *branch-name* to be a candidate key for *branch*. If, at the time we enter the **create unique index** command, *branch-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>

12.9 Multiple-Key Access

Until now, we have assumed implicitly that only one index (or hash table) 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.

12.9.1 Using Multiple Single-Key Indices

Assume that the *account* file has two indices: one for *branch-name* and one for *balance*. Consider the following query: "Find all account numbers at the Perryridge branch with balances equal to $1000." We write

select *loan-number*
from *account*
where *branch-name* = "Perryridge" **and** *balance* = 1000

There are three strategies possible for processing this query:

1. Use the index on *branch-name* to find all records pertaining to the Perryridge branch. Examine each such record to see whether *balance* = 1000.
2. Use the index on *balance* to find all records pertaining to accounts with balances of $1000. Examine each such record to see whether *branch-name* = "Perryridge."
3. Use the index on *branch-name* to find *pointers* to all records pertaining to the Perryridge branch. Also, use the index on *balance* to find pointers to all records

pertaining to accounts with a balance of $1000. Take the intersection of these two sets of pointers. Those pointers that are in the intersection point to records pertaining to both Perryridge and accounts with a balance of $1000.

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 Perryridge branch.
- There are many records pertaining to accounts with a balance of $1000.
- There are only a few records pertaining to *both* the Perryridge branch and accounts with a balance of $1000.

If these conditions hold, we must scan a large number of pointers to produce a small result. An index structure called a "bitmap index" greatly speeds up the intersection operation used in the third strategy. Bitmap indices are outlined in Section 12.9.4.

12.9.2 Indices on Multiple Keys

An alternative strategy for this case is to create and use an index on a search key (*branch-name*, *balance*)—that is, the search key consisting of the branch name concatenated with the account balance. 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.

The use of an ordered-index structure on multiple attributes has a few shortcomings. As an illustration, consider the query

select *loan-number*
from *account*
where *branch-name* $<$ "Perryridge" **and** *balance* $= 1000$

We can answer this query by using an ordered index on the search key (*branch-name*, *balance*): For each value of *branch-name* that is less than "Perryridge" in alphabetic order, the system locates records with a *balance* value of 1000. 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 one is that the condition on *branch-name* is a comparison condition, rather than an equality condition.

To speed the processing of general multiple search-key queries (which can involve one or more comparison operations), we can use several special structures. We shall consider the *grid file* in Section 12.9.3. 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 23.

12.9.3 Grid Files

Figure 12.32 shows part of a **grid file** for the search keys *branch-name* and *balance* on the *account* file. The two-dimensional array in the figure is called the *grid array*, and the one-dimensional arrays are called *linear scales*. The grid file has a single grid array, and one linear scale for each search-key attribute.

Search keys are mapped to cells in this way. Each cell in the grid array has a pointer to a bucket that contains the search-key values and pointers to records. Only some of the buckets and pointers from the cells are shown in the figure. To conserve space, multiple elements of the array can point to the same bucket. The dotted boxes in the figure indicate which cells point to the same bucket.

Suppose that we want to insert in the grid-file index a record whose search-key value is ("Brighton", 500000). To find the cell to which the key is mapped, we independently locate the row and column to which the cell belongs.

We first use the linear scales on *branch-name* to locate the row of the cell to which the search key maps. To do so, we search the array to find the least element that is greater than "Brighton". In this case, it is the first element, so the search key maps to the row marked 0. If it were the ith element, the search key would map to row $i - 1$. If the search key is greater than or equal to all elements in the linear scale, it maps to

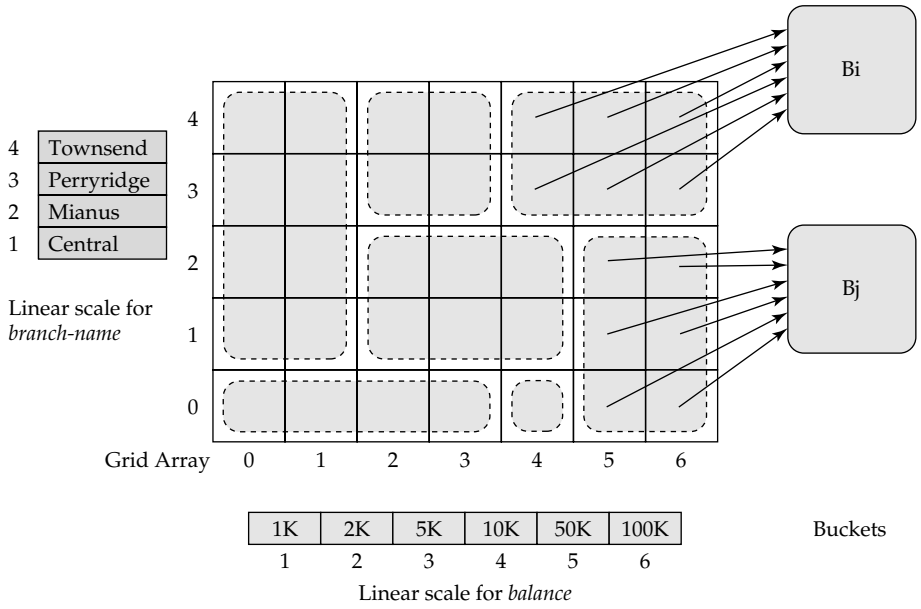


Figure 12.32 Grid file on keys *branch-name* and *balance* of the *account* file.

the final row. Next, we use the linear scale on *balance* to find out similarly to which column the search key maps. In this case, the balance 500000 maps to column 6.

Thus, the search-key value ("Brighton", 500000) maps to the cell in row 0, column 6. Similarly, ("Downtown", 60000) would map to the cell in row 1 column 5. Both cells point to the same bucket (as indicated by the dotted box), so, in both cases, the system stores the search-key values and the pointer to the record in the bucket labeled B_j in the figure.

To perform a lookup to answer our example query, with the search condition of

$$\textit{branch-name} < \text{"Perryridge"} \textbf{ and } \textit{balance} = 1000$$

we find all rows that can contain branch names less than "Perryridge", using the linear scale on *branch-name*. In this case, these rows are 0, 1, and 2. Rows 3 and beyond contain branch names greater than or equal to "Perryridge". Similarly, we find that only column 1 can contain a *balance* value of 1000. In this case, only column 1 satisfies this condition. Thus, only the cells in column 1, rows 0, 1, and 2, can contain entries that satisfy the search condition.

We therefore look up all entries in the buckets pointed to from these three cells. In this case, there are only two buckets, since two of the cells point to the same bucket, as indicated by the dotted boxes in the figure. The buckets may contain some search keys that do not satisfy the required condition, so each search key in the buckets must be tested again to see whether it satisfies the search condition. We have to examine only a small number of buckets, however, to answer this query.

We must choose the linear scales in such a way that the records are uniformly distributed across the cells. When a bucket—call it A—becomes full and an entry has to be inserted in it, the system allocates an extra bucket, B. If more than one cell points to A, the system changes the cell pointers so that some point to A and others to B. The entries in bucket A and the new entry are then redistributed between A and B according to the cells to which they map. If only one cell points to bucket A, B becomes an overflow bucket for A. To improve performance in such a situation, we must reorganize the grid file, with an expanded grid array and expanded linear scales. The process is much like the expansion of the bucket address table in extensible hashing, and is left for you to do as an exercise.

It is conceptually simple to extend the grid-file approach to any number of search keys. If we want our structure to be used for queries on n keys, we construct an n-dimensional grid array with n linear scales.

The grid structure is suitable also for queries involving one search key. Consider this query:

select *
from *account*
where *branch-name* = "Perryridge"

The linear scale on *branch-name* tells us that only cells in row 3 can satisfy this condition. Since there is no condition on *balance*, we examine all buckets pointed to by cells in row 3 to find entries pertaining to Perryridge. Thus, we can use a grid-file index on

two search keys to answer queries on either search key by itself, as well as to answer queries on both search keys. Thus, a single grid-file index can serve the role of three separate indices. If each index were maintained separately, the three together would occupy more space, and the cost of updating them would be high.

Grid files provide a significant decrease in processing time for multiple-key queries. However, they impose a space overhead (the grid directory can become large), as well as a performance overhead on record insertion and deletion. Further, it is hard to choose partitioning ranges for the keys such that the distribution of records is uniform. If insertions to the file are frequent, reorganization will have to be carried out periodically, and that can have a high cost.

12.9.4 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 *customer-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.

12.9.4.1 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 ith 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 ith 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 12.33 shows an example of bitmap indices on a relation *customer-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.

record number	*name*	*gender*	*address*	*income-level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income-level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Figure 12.33 Bitmap indices on relation *customer-info*.

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 $10,000 - 19,999$. This query can be expressed as $\sigma_{gender=\text{f} \wedge income\text{-}level=L2}(r)$. 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 ith bit of the two bitmaps are both 1, and has a value 0 otherwise. In the example in Figure 12.33, the intersection of the bitmap for $gender = \text{f}$ (01101) and the bitmap for $income\text{-}level = L1$ (10100) gives the bitmap 00100.

Since the first attribute can take 2 values, and the second can take 5 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 8 values, a bitmap index on attribute A would consist of 8 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 will see the need for existence bitmaps in Section 12.9.4.2. 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.

12.9.4.2 Efficient Implementation of Bitmap Operations

We can compute the intersection of two bitmaps easily by using a **for** loop: the ith iteration of the loop computes the **and** of the ith 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 Kbytes. 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 like 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*.[1]

Counting the number of bits that are 1 in a bitmap can be done fast by a clever technique. We can maintain an array with 256 entries, where the ith entry stores the

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

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 would be $\frac{1}{8}$ of the number of tuples, and thus the counting process is very efficient. A large array (using $2^{16} = 65536$ entries), indexed by pairs of bytes, would give even higher speedup, but at a higher storage cost.

12.9.4.3 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.

12.10 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 *primary index*. The other indices are called *secondary indices*. Secondary indices improve the performance of queries that use search keys other than the primary one. 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.
- We can also use hashing to create secondary indices; such indices are called *hash indices*. For notational convenience, we assume hash file organizations have an implicit hash index on the search key used for hashing.
- Ordered indices such as B^+-trees and hash indices can be used for selections based on equality conditions involving single attributes. When multiple

attributes are involved in a selection condition, we can intersect record identifiers retrieved from multiple indices.

- Grid files provide a general means of indexing on multiple attributes.
- Bitmap indices provide a very compact representation for indexing attributes with very few distinct values. Intersection operations are extremely fast on bitmaps, making them ideal for supporting queries on multiple attributes.

Review Terms

- Access types
- Access time
- Insertion time
- Deletion time
- Space overhead
- Ordered index
- Primary index
- Clustering index
- Secondary index
- Nonclustering index
- Index-sequential file
- Index record/entry
- Dense index
- Sparse index
- Multilevel index
- Sequential scan
- B^+-Tree index
- Balanced tree
- B^+-Tree file organization
- B-Tree index
- Static hashing
- Hash file organization
- Hash index
- Bucket
- Hash function
- Bucket overflow
- Skew
- Closed hashing
- Dynamic hashing
- Extendable hashing
- Multiple-key access
- Indices on multiple keys
- Grid files
- Bitmap index
- Bitmap operations
  - ☐ Intersection
  - ☐ Union
  - ☐ Complement
  - ☐ Existence bitmap

Exercises

12.1 When is it preferable to use a dense index rather than a sparse index? Explain your answer.

12.2 Since indices speed query processing, why might they not be kept on several search keys? List as many reasons as possible.

12.3 What is the difference between a primary index and a secondary index?

12.4 Is it possible in general to have two primary indices on the same relation for different search keys? Explain your answer.

12.5 Construct a B^+-tree for the following set of key values:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Assume that the tree is initially empty and values are added in ascending order. Construct B^+-trees for the cases where the number of pointers that will fit in one node is as follows:

a. Four
b. Six
c. Eight

12.6 For each B^+-tree of Exercise 12.5, show the steps involved in the following queries:

a. Find records with a search-key value of 11.
b. Find records with a search-key value between 7 and 17, inclusive.

12.7 For each B^+-tree of Exercise 12.5, show the form of the tree after each of the following series of operations:

a. Insert 9.
b. Insert 10.
c. Insert 8.
d. Delete 23.
e. Delete 19.

12.8 Consider the modified redistribution scheme for B^+-trees described in page 463. What is the expected height of the tree as a function of n?

12.9 Repeat Exercise 12.5 for a B-tree.

12.10 Explain the distinction between closed and open hashing. Discuss the relative merits of each technique in database applications.

12.11 What are the causes of bucket overflow in a hash file organization? What can be done to reduce the occurrence of bucket overflows?

12.12 Suppose that we are using extendable hashing on a file that contains records with the following search-key values:

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

Show the extendable hash structure for this file if the hash function is $h(x) = x$ mod 8 and buckets can hold three records.

12.13 Show how the extendable hash structure of Exercise 12.12 changes as the result of each of the following steps:

a. Delete 11.
b. Delete 31.
c. Insert 1.
d. Insert 15.

12.14 Give pseudocode for deletion of entries from an extendable hash structure, including details of when and how to coalesce buckets. Do not bother about reducing the size of the bucket address table.

12.15 Suggest an efficient way to test if the bucket address table in extendable hashing can be reduced in size, by storing an extra count with the bucket address table. Give details of how the count should be maintained when buckets are split, coalesced or deleted.

(Note: Reducing the size of the bucket address table is an expensive operation, and subsequent inserts may cause the table to grow again. Therefore, it is best not to reduce the size as soon as it is possible to do so, but instead do it only if the number of index entries becomes small compared to the bucket address table size.)

12.16 Why is a hash structure not the best choice for a search key on which range queries are likely?

12.17 Consider a grid file in which we wish to avoid overflow buckets for performance reasons. In cases where an overflow bucket would be needed, we instead reorganize the grid file. Present an algorithm for such a reorganization.

12.18 Consider the *account* relation shown in Figure 12.25.

a. Construct a bitmap index on the attributes *branch-name* and *balance*, dividing *balance* values into 4 ranges: below 250, 250 to below 500, 500 to below 750, and 750 and above.

b. Consider a query that requests all accounts in Downtown with a balance of 500 or more. Outline the steps in answering the query, and show the final and intermediate bitmaps constructed to answer the query.

12.19 Show how to compute existence bitmaps from other bitmaps. Make sure that your technique works even in the presence of null values, by using a bitmap for the value *null*.

12.20 How does data encryption affect index schemes? In particular, how might it affect schemes that attempt to store data in sorted order?

Bibliographical Notes

Discussions of the basic data structures in indexing and hashing can be found in Cormen et al. [1990]. B-tree indices were first introduced in Bayer [1972] and Bayer and McCreight [1972]. B^+-trees are discussed in Comer [1979], Bayer and Unterauer [1977] and Knuth [1973]. The bibliographic notes in Chapter 16 provides references to research on allowing concurrent accesses and updates on B^+-trees. Gray and Reuter [1993] provide a good description of issues in the implementation of B^+-trees.

Several alternative tree and treelike search structures have been proposed. **Tries** are trees whose structure is based on the "digits" of keys (for example, a dictionary thumb index, which has one entry for each letter). Such trees may not be balanced in the sense that B^+-trees are. Tries are discussed by Ramesh et al. [1989], Orenstein

[1982], Litwin [1981] and Fredkin [1960]. Related work includes the digital B-trees of Lomet [1981].

Knuth [1973] analyzes a large number of different hashing techniques. Several dynamic hashing schemes exist. Extendable hashing was introduced by Fagin et al. [1979]. Linear hashing was introduced by Litwin [1978] and Litwin [1980]; Larson [1982] presents a performance analysis of linear hashing. Ellis [1987] examined concurrency with linear hashing. Larson [1988] presents a variant of linear hashing. Another scheme, called dynamic hashing, was proposed by Larson [1978]. An alternative given by Ramakrishna and Larson [1989] allows retrieval in a single disk access at the price of a high overhead for a small fraction of database modifications. Partitioned hashing is an extension of hashing to multiple attributes, and is covered in Rivest [1976], Burkhard [1976] and Burkhard [1979].

The grid file structure appears in Nievergelt et al. [1984] and Hinrichs [1985]. Bitmap indices, and variants called **bit-sliced indices** and **projection indices** are described in O'Neil and Quass [1997]. They were first introduced in the IBM Model 204 file manager on the AS 400 platform. They provide very large speedups on certain types of queries, and are today implemented on most database systems. Recent research on bitmap indices includes Wu and Buchmann [1998], Chan and Ioannidis [1998], Chan and Ioannidis [1999], and Johnson [1999a].

CHAPTER 13

Query Processing

Query processing refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

13.1 Overview

The steps involved in processing a query appear in Figure 13.1. The basic steps are

1. Parsing and translation
2. Optimization
3. Evaluation

Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but is ill-suited to be the system's internal representation of a query. A more useful internal representation is one based on the extended relational algebra.

Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expres-

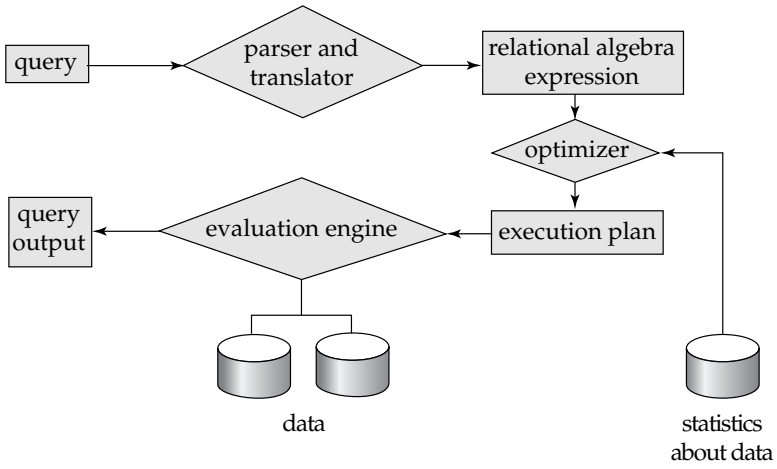


Figure 13.1 Steps in query processing.

sion that defines the view.[1] Most compiler texts cover parsing (see the bibliographical notes).

Given a query, there are generally a variety of methods for computing the answer. For example, we have seen that, in SQL, a query could be expressed in several different ways. Each SQL query can itself be translated into a relational-algebra expression in one of several ways. Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions. As an illustration, consider the query

select *balance*
from *account*
where *balance* < 2500

This query can be translated into either of the following relational-algebra expressions:

- $\sigma_{balance<2500}\ (\Pi_{balance}\ (account))$
- $\Pi_{balance}\ (\sigma_{balance<2500}\ (account))$

Further, we can execute each relational-algebra operation by one of several different algorithms. For example, to implement the preceding selection, we can search every tuple in *account* to find tuples with balance less than 2500. If a B^+-tree index is available on the attribute *balance*, we can use the index instead to locate the tuples.

To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to eval-

1. For materialized views, the expression defining the view has already been evaluated and stored. Therefore, the stored relation can be used, instead of uses of the view being replaced by the expression defining the view. Recursive views are handled differently, via a fixed-point procedure, as discussed in Section 5.2.6.

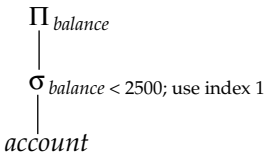


Figure 13.2 A query-evaluation plan.

uate each operation. Annotations may state the algorithm to be used for a specific operation, or the particular index or indices to use. A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**. A sequence of primitive operations that can be used to evaluate a query is a **query-execution plan** or **query-evaluation plan**. Figure 13.2 illustrates an evaluation plan for our example query, in which a particular index (denoted in the figure as "index 1") is specified for the selection operation. The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

The different evaluation plans for a given query can have different costs. We do not expect users to write their queries in a way that suggests the most efficient evaluation plan. Rather, it is the responsibility of the system to construct a query-evaluation plan that minimizes the cost of query evaluation. Chapter 14 describes query optimization in detail.

Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.

The sequence of steps already described for processing a query is representative; not all databases exactly follow those steps. For instance, instead of using the relational-algebra representation, several databases use an annotated parse-tree representation based on the structure of the given SQL query. However, the concepts that we describe here form the basis of query processing in databases.

In order to optimize a query, a query optimizer must know the cost of each operation. Although the exact cost is hard to compute, since it depends on many parameters such as actual memory available to the operation, it is possible to get a rough estimate of execution cost for each operation.

Section 13.2 outlines how we measure the cost of a query. Sections 13.3 through 13.6 cover the evaluation of individual relational-algebra operations. Several operations may be grouped together into a **pipeline**, in which each of the operations starts working on its input tuples even as they are being generated by another operation. In Section 13.7, we examine how to coordinate the execution of multiple operations in a query evaluation plan, in particular, how to use pipelined operations to avoid writing intermediate results to disk.

13.2 Measures of Query Cost

The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication (which we discuss later, in

Chapters 19 and 20). The response time for a query-evaluation plan (that is, the clock time required to execute the plan), assuming no other activity is going on on the computer, would account for all these costs, and could be used as a good measure of the cost of the plan.

In large database systems, however, disk accesses (which we measure as the number of transfers of blocks from disk) are usually the most important cost, since disk accesses are slow compared to in-memory operations. Moreover, CPU speeds have been improving much faster than have disk speeds. Thus, it is likely that the time spent in disk activity will continue to dominate the total time to execute a query. Finally, estimating the CPU time is relatively hard, compared to estimating the disk-access cost. Therefore, most people consider the disk-access cost a reasonable measure of the cost of a query-evaluation plan.

We use the *number of block transfers* from disk as a measure of the actual cost. To simplify our computation of disk-access cost, we assume that all transfers of blocks have the same cost. This assumption ignores the variance arising from rotational latency (waiting for the desired data to spin under the read–write head) and seek time (the time that it takes to move the head over the desired track or cylinder). To get more precise numbers, we need to distinguish between **sequential I/O**, where the blocks read are contiguous on disk, and **random I/O**, where the blocks are noncontiguous, and an extra seek cost must be paid for each disk I/O operation. We also need to distinguish between reads and writes of blocks, since it takes more time to write a block to disk than to read a block from disk. A more accurate measure would therefore estimate

1. The number of seek operations performed
2. The number of blocks read
3. The number of blocks written

and then add up these numbers after multiplying them by the average seek time, average transfer time for reading a block, and average transfer time for writing a block, respectively. Real-life query optimizers also take CPU costs into account when computing the cost of an operation. For simplicity we ignore these details, and leave it to you to work out more precise cost estimates for various operations.

The cost estimates we give ignore the cost of writing the final result of an operation back to disk. These are taken into account separately where required. The costs of all the algorithms that we consider depend on the size of the buffer in main memory. In the best case, all data can be read into the buffers, and the disk does not need to be accessed again. In the worst case, we assume that the buffer can hold only a few blocks of data—approximately one block per relation. When presenting cost estimates, we generally assume the worst case.

13.3 Selection Operation

In query processing, the **file scan** is the lowest-level operator to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition.

In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.

13.3.1 Basic Algorithms

Consider a selection operation on a relation whose tuples are stored together in one file. Two scan algorithms to implement the selection operation are:

- **A1** (**linear search**). In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. For a selection on a key attribute, the system can terminate the scan if the required record is found, without looking at the other records of the relation.

  The cost of linear search, in terms of number of I/O operations, is b_r, where b_r denotes the number of blocks in the file. Selections on key attributes have an average cost of $b_r/2$, but still have a worst-case cost of b_r.

  Although it may be slower than other algorithms for implementing selection, the linear search algorithm can be applied to any file, regardless of the ordering of the file, or the availability of indices, or the nature of the selection operation. The other algorithms that we shall study are not applicable in all cases, but when applicable they are generally faster than linear search.

- **A2** (**binary search**). If the file is ordered on an attribute, and the selection condition is an equality comparison on the attribute, we can use a binary search to locate records that satisfy the selection. The system performs the binary search on the blocks of the file.

  The number of blocks that need to be examined to find a block containing the required records is $\lceil \log_2(b_r) \rceil$, where b_r denotes the number of blocks in the file. If the selection is on a nonkey attribute, more than one block may contain required records, and the cost of reading the extra blocks has to be added to the cost estimate. We can estimate this number by estimating the size of the selection result (which we cover in Section 14.2), and dividing it by the average number of records that are stored per block of the relation.

13.3.2 Selections Using Indices

Index structures are referred to as **access paths**, since they provide a path through which data can be located and accessed. In Chapter 12, we pointed out that it is efficient to read the records of a file in an order corresponding closely to physical order. Recall that a *primary index* is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file. An index that is not a primary index is called a *secondary index*.

Search algorithms that use an index are referred to as **index scans**. Ordered indices, such as B^+-trees, also permit access to tuples in a sorted order, which is useful for implementing range queries. Although indices can provide fast, direct, and ordered access, they impose the overhead of access to those blocks containing the index. We use the selection predicate to guide us in the choice of the index to use in processing the query. Search algorithms that use an index are:

- **A3** (**primary index, equality on key**). For an equality comparison on a key attribute with a primary index, we can use the index to retrieve a single record that satisfies the corresponding equality condition.

  If a B$^+$-tree is used, the cost of the operation, in terms of I/O operations, is equal to the height of the tree plus one I/O to fetch the record.

- **A4** (**primary index, equality on nonkey**). We can retrieve multiple records by using a primary index when the selection condition specifies an equality comparison on a nonkey attribute, A. The only difference from the previous case is that multiple records may need to be fetched. However, the records would be stored consecutively in the file since the file is sorted on the search key.

  The cost of the operation is proportional to the height of the tree, plus the number of blocks containing records with the specified search key.

- **A5** (**secondary index, equality**). Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may get retrieved if the indexing field is not a key.

  In the first case, only one record is retrieved, and the cost is equal to the height of the tree plus one I/O operation to fetch the record. In the second case each record may be resident on a different block, which may result in one I/O operation per retrieved record. The cost could become even worse than that of linear search if a large number of records are retrieved.

If B$^+$-tree file organizations are used to store relations, records may be moved between blocks when leaf nodes are split or merged, and when records are redistributed. If secondary indices store pointers to records' physical location, the pointers will have to be updated when records are moved. In some systems, such as Compaq's Non-Stop SQL System, the secondary indices instead store the key value in the B$^+$-tree file organization. Accessing a record through a secondary index is then even more expensive since a search has to be performed on the B$^+$-tree used in the file organization. The cost formulae described for secondary indices will have to be modified appropriately if such indices are used.

13.3.3 Selections Involving Comparisons

Consider a selection of the form $\sigma_{A \leq v}(r)$. We can implement the selection either by using a linear or binary search or by using indices in one of the following ways:

- **A6** (**primary index, comparison**). A primary ordered index (for example, a primary B$^+$-tree index) can be used when the selection condition is a comparison. For comparison conditions of the form $A > v$ or $A \geq v$, a primary index on A can be used to direct the retrieval of tuples, as follows. For $A \geq v$, we look up the value v in the index to find the first tuple in the file that has a value of $A = v$. A file scan starting from that tuple up to the end of the file returns

all tuples that satisfy the condition. For $A > v$, the file scan starts with the first tuple such that $A > v$.

For comparisons of the form $A < v$ or $A \leq v$, an index lookup is not required. For $A < v$, we use a simple file scan starting from the beginning of the file, and continuing up to (but not including) the first tuple with attribute $A = v$. The case of $A \leq v$ is similar, except that the scan continues up to (but not including) the first tuple with attribute $A > v$. In either case, the index is not useful.

- **A7** (**secondary index, comparison**). We can use a secondary ordered index to guide retrieval for comparison conditions involving $<, \leq, \geq$, or $>$. The lowest-level index blocks are scanned, either from the smallest value up to v (for $<$ and $\leq$), or from v up to the maximum value (for $>$ and $\geq$).

  The secondary index provides pointers to the records, but to get the actual records we have to fetch the records by using the pointers. This step may require an I/O operation for each record fetched, since consecutive records may be on different disk blocks. If the number of retrieved records is large, using the secondary index may be even more expensive than using linear search. Therefore the secondary index should be used only if very few records are selected.

13.3.4 Implementation of Complex Selections

So far, we have considered only simple selection conditions of the form $A \; op \; B$, where *op* is an equality or comparison operation. We now consider more complex selection predicates.

- **Conjunction:** A *conjunctive selection* is a selection of the form

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

- **Disjunction:** A *disjunctive selection* is a selection of the form

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

  A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θ_i.

- **Negation:** The result of a selection $\sigma_{\neg\theta}(r)$ is the set of tuples of r for which the condition θ evaluates to false. In the absence of nulls, this set is simply the set of tuples that are not in $\sigma_\theta(r)$.

We can implement a selection operation involving either a conjunction or a disjunction of simple conditions by using one of the following algorithms:

- **A8** (**conjunctive selection using one index**). We first determine whether an access path is available for an attribute in one of the simple conditions. If one

is, one of the selection algorithms A2 through A7 can retrieve records satisfying that condition. We complete the operation by testing, in the memory buffer, whether or not each retrieved record satisfies the remaining simple conditions.

To reduce the cost, we choose a θ_i and one of algorithms A1 through A7 for which the combination results in the least cost for $\sigma_{\theta_i}(r)$. The cost of algorithm A8 is given by the cost of the chosen algorithm.

- **A9** (**conjunctive selection using composite index**). An appropriate *composite index* (that is, an index on multiple attributes) may be available for some conjunctive selections. If the selection specifies an equality condition on two or more attributes, and a composite index exists on these combined attribute fields, then the index can be searched directly. The type of index determines which of algorithms A3, A4, or A5 will be used.

- **A10** (**conjunctive selection by intersection of identifiers**). Another alternative for implementing conjunctive selection operations involves the use of record pointers or record identifiers. This algorithm requires indices with record pointers, on the fields involved in the individual conditions. The algorithm scans each index for pointers to tuples that satisfy an individual condition. The intersection of all the retrieved pointers is the set of pointers to tuples that satisfy the conjunctive condition. The algorithm then uses the pointers to retrieve the actual records. If indices are not available on all the individual conditions, then the algorithm tests the retrieved records against the remaining conditions.

  The cost of algorithm **A10** is the sum of the costs of the individual index scans, plus the cost of retrieving the records in the intersection of the retrieved lists of pointers. This cost can be reduced by sorting the list of pointers and retrieving records in the sorted order. Thereby, (1) all pointers to records in a block come together, hence all selected records in the block can be retrieved using a single I/O operation, and (2) blocks are read in sorted order, minimizing disk arm movement. Section 13.4 describes sorting algorithms.

- **A11** (**disjunctive selection by union of identifiers**). If access paths are available on all the conditions of a disjunctive selection, each index is scanned for pointers to tuples that satisfy the individual condition. The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy the disjunctive condition. We then use the pointers to retrieve the actual records.

  However, if even one of the conditions does not have an access path, we will have to perform a linear scan of the relation to find tuples that satisfy the condition. Therefore, if there is even one such condition in the disjunct, the most efficient access method is a linear scan, with the disjunctive condition tested on each tuple during the scan.

The implementation of selections with negation conditions is left to you as an exercise (Exercise 13.10).

13.4 Sorting

Sorting of data plays an important role in database systems for two reasons. First, SQL queries can specify that the output be sorted. Second, and equally important for query processing, several of the relational operations, such as joins, can be implemented efficiently if the input relations are first sorted. Thus, we discuss sorting here before discussing the join operation in Section 13.5.

We can sort a relation by building an index on the sort key, and then using that index to read the relation in sorted order. However, such a process orders the relation only *logically*, through an index, rather than *physically*. Hence, the reading of tuples in the sorted order may lead to a disk access for each record, which can be very expensive, since the number of records can be much larger than the number of blocks. For this reason, it may be desirable to order the records physically.

The problem of sorting has been studied extensively, both for relations that fit entirely in main memory, and for relations that are bigger than memory. In the first case, standard sorting techniques such as quick-sort can be used. Here, we discuss how to handle the second case.

Sorting of relations that do not fit in memory is called **external sorting**. The most commonly used technique for external sorting is the **external sort–merge** algorithm. We describe the external sort–merge algorithm next. Let M denote the number of page frames in the main-memory buffer (the number of disk blocks whose contents can be buffered in main memory).

1. In the first stage, a number of sorted **runs** are created; each run is sorted, but contains only some of the records of the relation.

   ```
   i = 0;
   repeat
       read M blocks of the relation, or the rest of the relation,
            whichever is smaller;
       sort the in-memory part of the relation;
       write the sorted data to run file R_i;
       i = i + 1;
   until the end of the relation
   ```

2. In the second stage, the runs are *merged*. Suppose, for now, that the total number of runs, N, is less than M, so that we can allocate one page frame to each run and have space left to hold one page of output. The merge stage operates as follows:

   ```
   read one block of each of the N files R_i into a buffer page in memory;
   repeat
       choose the first tuple (in sort order) among all buffer pages;
       write the tuple to the output, and delete it from the buffer page;
       if the buffer page of any run R_i is empty and not end-of-file(R_i)
           then read the next block of R_i into the buffer page;
   until all buffer pages are empty
   ```

The output of the merge stage is the sorted relation. The output file is buffered to reduce the number of disk write operations. The preceding merge operation is a generalization of the two-way merge used by the standard in-memory sort–merge algorithm; it merges N runs, so it is called an **N-way merge**.

In general, if the relation is much larger than memory, there may be M or more runs generated in the first stage, and it is not possible to allocate a page frame for each run during the merge stage. In this case, the merge operation proceeds in multiple passes. Since there is enough memory for $M - 1$ input buffer pages, each merge can take $M - 1$ runs as input.

The initial *pass* functions in this way: It merges the first $M - 1$ runs (as described in item 2 above) to get a single run for the next pass. Then, it merges the next $M - 1$ runs similarly, and so on, until it has processed all the initial runs. At this point, the number of runs has been reduced by a factor of $M - 1$. If this reduced number of runs is still greater than or equal to M, another pass is made, with the runs created by the first pass as input. Each pass reduces the number of runs by a factor of $M - 1$. The passes repeat as many times as required, until the number of runs is less than M; a final pass then generates the sorted output.

Figure 13.3 illustrates the steps of the external sort–merge for an example relation. For illustration purposes, we assume that only one tuple fits in a block ($f_r = 1$), and we assume that memory holds at most three page frames. During the merge stage, two page frames are used for input and one for output.

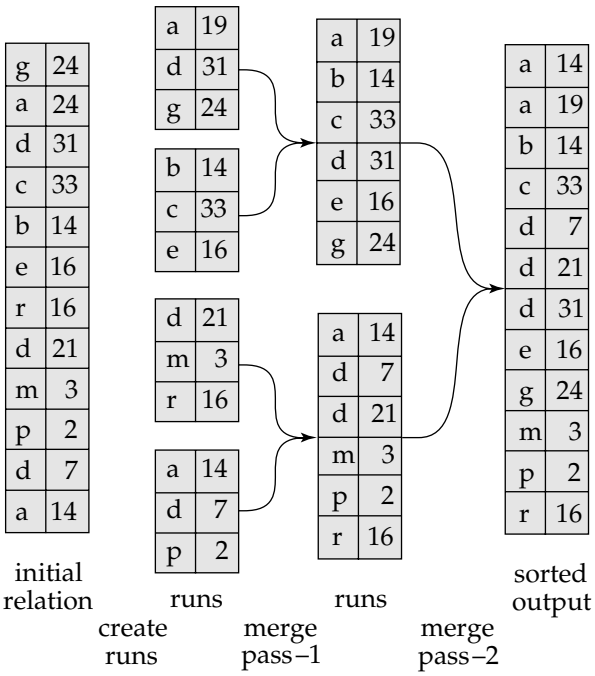


Figure 13.3 External sorting using sort–merge.

We compute how many block transfers are required for the external sort merge in this way: Let b_r denote the number of blocks containing records of relation r. The first stage reads every block of the relation and writes them out again, giving a total of $2b_r$ disk accesses. The initial number of runs is $\lceil b_r/M \rceil$. Since the number of runs decreases by a factor of $M-1$ in each merge pass, the total number of merge passes required is $\lceil \log_{M-1}(b_r/M) \rceil$. Each of these passes reads every block of the relation once and writes it out once, with two exceptions. First, the final pass can produce the sorted output without writing its result to disk. Second, there may be runs that are not read in or written out during a pass—for example, if there are M runs to be merged in a pass, $M-1$ are read in and merged, and one run is not accessed during the pass. Ignoring the (relatively small) savings due to the latter effect, the total number of disk accesses for external sorting of the relation is

$$b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

Applying this equation to the example in Figure 13.3, we get a total of $12*(4+1) = 60$ block transfers, as you can verify from the figure. Note that this value does not include the cost of writing out the final result.

13.5 Join Operation

In this section, we study several algorithms for computing the join of relations, and we analyze their respective costs.

We use the term **equi-join** to refer to a join of the form $r \bowtie_{r.A=s.B} s$, where A and B are attributes or sets of attributes of relations r and s respectively.

We use as a running example the expression

$$\textit{depositor} \bowtie \textit{customer}$$

We assume the following information about the two relations:

- Number of records of *customer*: $n_{customer} = 10,000$.
- Number of blocks of *customer*: $b_{customer} = 400$.
- Number of records of *depositor*: $n_{depositor} = 5000$.
- Number of blocks of *depositor*: $b_{depositor} = 100$.

13.5.1 Nested-Loop Join

Figure 13.4 shows a simple algorithm to compute the theta join, $r \bowtie_\theta s$, of two relations r and s. This algorithm is called the **nested-loop join** algorithm, since it basically consists of a pair of nested **for** loops. Relation r is called the **outer relation** and relation s the **inner relation** of the join, since the loop for r encloses the loop for s. The algorithm uses the notation $t_r \cdot t_s$, where t_r and t_s are tuples; $t_r \cdot t_s$ denotes the tuple constructed by concatenating the attribute values of tuples t_r and t_s.

Like the linear file-scan algorithm for selection, the nested-loop join algorithm requires no indices, and it can be used regardless of what the join condition is. Extending the algorithm to compute the natural join is straightforward, since the natural

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r · t_s to the result.
    end
end
```

Figure 13.4 Nested-loop join.

join can be expressed as a theta join followed by elimination of repeated attributes by a projection. The only change required is an extra step of deleting repeated attributes from the tuple $t_r \cdot t_s$, before adding it to the result.

The nested-loop join algorithm is expensive, since it examines every pair of tuples in the two relations. Consider the cost of the nested-loop join algorithm. The number of pairs of tuples to be considered is $n_r * n_s$, where n_r denotes the number of tuples in r, and n_s denotes the number of tuples in s. For each record in r, we have to perform a complete scan on s. In the worst case, the buffer can hold only one block of each relation, and a total of $n_r * b_s + b_r$ block accesses would be required, where b_r and b_s denote the number of blocks containing tuples of r and s respectively. In the best case, there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once; hence, only $b_r + b_s$ block accesses would be required.

If one of the relations fits entirely in main memory, it is beneficial to use that relation as the inner relation, since the inner relation would then be read only once. Therefore, if s is small enough to fit in main memory, our strategy requires only a total $b_r + b_s$ accesses—the same cost as that for the case where both relations fit in memory.

Now consider the natural join of *depositor* and *customer*. Assume for now that we have no indices whatsoever on either relation, and that we are not willing to create any index. We can use the nested loops to compute the join; assume that *depositor* is the outer relation and *customer* is the inner relation in the join. We will have to examine $5000 * 10000 = 50 * 10^6$ pairs of tuples. In the worst case, the number of block accesses is $5000 * 400 + 100 =$ 2,000,100. In the best-case scenario, however, we can read both relations only once, and perform the computation. This computation requires at most $100 + 400 = 500$ block accesses—a significant improvement over the worst-case scenario. If we had used *customer* as the relation for the outer loop and *depositor* for the inner loop, the worst-case cost of our final strategy would have been lower: $10000 * 100 + 400 =$ 1,000,400.

13.5.2 Block Nested-Loop Join

If the buffer is too small to hold either relation entirely in memory, we can still obtain a major saving in block accesses if we process the relations on a per-block basis, rather than on a per-tuple basis. Figure 13.5 shows **block nested-loop join**, which is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                test pair (t_r, t_s) to see if they satisfy the join condition
                if they do, add t_r · t_s to the result.
            end
        end
    end
end
```

Figure 13.5 Block nested-loop join.

is paired with every tuple in the other block, to generate all pairs of tuples. As before, all pairs of tuples that satisfy the join condition are added to the result.

The primary difference in cost between the block nested-loop join and the basic nested-loop join is that, in the worst case, each block in the inner relation s is read only once for each *block* in the outer relation, instead of once for each *tuple* in the outer relation. Thus, in the worst case, there will be a total of $b_r * b_s + b_r$ block accesses, where b_r and b_s denote the number of blocks containing records of r and s respectively. Clearly, it is more efficient to use the smaller relation as the outer relation, in case neither of the relations fits in memory. In the best case, there will be $b_r + b_s$ block accesses.

Now return to our example of computing *depositor* $\bowtie$ *customer*, using the block nested-loop join algorithm. In the worst case we have to read each block of *customer* once for each block of *depositor*. Thus, in the worst case, a total of $100 * 400 + 100 = 40,100$ block accesses are required. This cost is a significant improvement over the $5000 * 400 + 100 = 2,000,100$ block accesses needed in the worst case for the basic nested-loop join. The number of block accesses in the best case remains the same—namely, $100 + 400 = 500$.

The performance of the nested-loop and block nested-loop procedures can be further improved:

- If the join attributes in a natural join or an equi-join form a key on the inner relation, then for each outer relation tuple the inner loop can terminate as soon as the first match is found.
- In the block nested-loop algorithm, instead of using disk blocks as the blocking unit for the outer relation, we can use the biggest size that can fit in memory, while leaving enough space for the buffers of the inner relation and the output. In other words, if memory has M blocks, we read in $M - 2$ blocks of the outer relation at a time, and when we read each block of the inner relation we join it with all the $M - 2$ blocks of the outer relation. This change reduces the number of scans of the inner relation from b_r to $\lceil b_r/(M-2) \rceil$, where b_r is the number of blocks of the outer relation. The total cost is then $\lceil b_r/(M-2) \rceil * b_s + b_r$.

- We can scan the inner loop alternately forward and backward. This scanning method orders the requests for disk blocks so that the data remaining in the buffer from the previous scan can be reused, thus reducing the number of disk accesses needed.
- If an index is available on the inner loop's join attribute, we can replace file scans with more efficient index lookups. Section 13.5.3 describes this optimization.

13.5.3 Indexed Nested-Loop Join

In a nested-loop join (Figure 13.4), if an index is available on the inner loop's join attribute, index lookups can replace file scans. For each tuple t_r in the outer relation r, the index is used to look up tuples in s that will satisfy the join condition with tuple t_r.

This join method is called an **indexed nested-loop join**; it can be used with existing indices, as well as with temporary indices created for the sole purpose of evaluating the join.

Looking up tuples in s that will satisfy the join conditions with a given tuple t_r is essentially a selection on s. For example, consider *depositor* $\bowtie$ *customer*. Suppose that we have a *depositor* tuple with *customer-name* "John". Then, the relevant tuples in s are those that satisfy the selection "*customer-name* $=$ John".

The cost of an indexed nested-loop join can be computed as follows. For each tuple in the outer relation r, a lookup is performed on the index for s, and the relevant tuples are retrieved. In the worst case, there is space in the buffer for only one page of r and one page of the index. Then, b_r disk accesses are needed to read relation r, where b_r denotes the number of blocks containing records of r. For each tuple in r, we perform an index lookup on s. Then, the cost of the join can be computed as $b_r + n_r * c$, where n_r is the number of records in relation r, and c is the cost of a single selection on s using the join condition. We have seen in Section 13.3 how to estimate the cost of a single selection algorithm (possibly using indices); that estimate gives us the value of c.

The cost formula indicates that, if indices are available on both relations r and s, it is generally most efficient to use the one with fewer tuples as the outer relation.

For example, consider an indexed nested-loop join of *depositor* $\bowtie$ *customer*, with *depositor* as the outer relation. Suppose also that *customer* has a primary B$^+$-tree index on the join attribute *customer-name*, which contains 20 entries on an average in each index node. Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data. Since $n_{depositor}$ is 5000, the total cost is $100 + 5000 * 5 = 25,100$ disk accesses. This cost is lower than the $40,100$ accesses needed for a block nested-loop join.

13.5.4 Merge Join

The **merge join** algorithm (also called the **sort–merge join** algorithm) can be used to compute natural joins and equi-joins. Let $r(R)$ and $s(S)$ be the relations whose natural join is to be computed, and let $R \cap S$ denote their common attributes. Suppose

```
pr := address of first tuple of r;
ps := address of first tuple of s;
while (ps ≠ null and pr ≠ null) do
    begin
        t_s := tuple to which ps points;
        S_s := {t_s};
        set ps to point to next tuple of s;
        done := false;
        while (not done and ps ≠ null) do
            begin
                t_s' := tuple to which ps points;
                if (t_s'[JoinAttrs] = t_s[JoinAttrs])
                    then begin
                            S_s := S_s ∪ {t_s'};
                            set ps to point to next tuple of s;
                         end
                    else done := true;
            end
        t_r := tuple to which pr points;
        while (pr ≠ null and t_r[JoinAttrs] < t_s[JoinAttrs]) do
            begin
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
        while (pr ≠ null and t_r[JoinAttrs] = t_s[JoinAttrs]) do
            begin
                for each t_s in S_s do
                    begin
                        add t_s ⋈ t_r to result ;
                    end
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
    end.
```

Figure 13.6 Merge join.

that both relations are sorted on the attributes $R \cap S$. Then, their join can be computed by a process much like the merge stage in the merge–sort algorithm.

Figure 13.6 shows the merge join algorithm. In the algorithm, *JoinAttrs* refers to the attributes in $R \cap S$, and $t_r \bowtie t_s$, where t_r and t_s are tuples that have the same values for *JoinAttrs*, denotes the concatenation of the attributes of the tuples, followed by projecting out repeated attributes. The merge join algorithm associates one pointer with each relation. These pointers point initially to the first tuple of the respective relations. As the algorithm proceeds, the pointers move through the relation. A group of tuples of one relation with the same value on the join attributes is read into S_s.

The algorithm in Figure 13.6 *requires* that every set of tuples S_s fit in main memory; we shall look at extensions of the algorithm to avoid this requirement later in this section. Then, the corresponding tuples (if any) of the other relation are read in, and are processed as they are read.

Figure 13.7 shows two relations that are sorted on their join attribute $a1$. It is instructive to go through the steps of the merge join algorithm on the relations shown in the figure.

Since the relations are in sorted order, tuples with the same value on the join attributes are in consecutive order. Thereby, each tuple in the sorted order needs to be read only once, and, as a result, each block is also read only once. Since it makes only a single pass through both files, the merge join method is efficient; the number of block accesses is equal to the sum of the number of blocks in both files, $b_r + b_s$.

If either of the input relations r and s is not sorted on the join attributes, they can be sorted first, and then the merge join algorithm can be used. The merge join algorithm can also be easily extended from natural joins to the more general case of equi-joins.

Suppose the merge join scheme is applied to our example of *depositor* $\bowtie$ *customer*. The join attribute here is *customer-name*. Suppose that the relations are already sorted on the join attribute *customer-name*. In this case, the merge join takes a total of $400 + 100 = 500$ block accesses. Suppose the relations are not sorted, and the memory size is the worst case of three blocks. Sorting *customer* takes $400 * (2\lceil \log_2(400/3) \rceil + 1)$, or 6800, block transfers, with 400 more transfers to write out the result. Similarly, sorting *depositor* takes $100 * (2\lceil \log_2(100/3) \rceil + 1)$, or 1300, transfers, with 100 more transfers to write it out. Thus, the total cost is 9100 block transfers if the relations are not sorted, and the memory size is just 3 blocks.

With a memory size of 25 blocks, sorting the relation *customer* takes a total of just $400 * (2\lceil \log_{24}(400/25) + 1) = 1200$ block transfers, while sorting *depositor* takes 300 block transfers. Adding the cost of writing out the sorted results and reading them back gives a total cost of 2500 block transfers if the relations are not sorted and the memory size is 25 blocks.

As mentioned earlier, the merge join algorithm of Figure 13.6 requires that the set S_s of all tuples with the same value for the join attributes must fit in main memory.

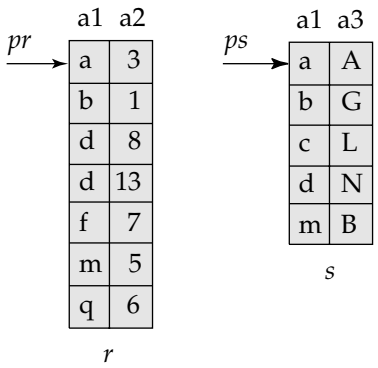


Figure 13.7 Sorted relations for merge join.

This requirement can usually be met, even if the relation s is large. If it cannot be met, a block nested-loop join must be performed between S_s and the tuples in r with the same values for the join attributes. The overall cost of the merge join increases as a result.

It is also possible to perform a variation of the merge join operation on unsorted tuples, if secondary indices exist on both join attributes. The algorithm scans the records through the indices, resulting in their being retrieved in sorted order. This variation presents a significant drawback, however, since records may be scattered throughout the file blocks. Hence, each tuple access could involve accessing a disk block, and that is costly.

To avoid this cost, we can use a hybrid merge–join technique, which combines indices with merge join. Suppose that one of the relations is sorted; the other is unsorted, but has a secondary B^+-tree index on the join attributes. The **hybrid merge–join algorithm** merges the sorted relation with the leaf entries of the secondary B^+-tree index. The result file contains tuples from the sorted relation and addresses for tuples of the unsorted relation. The result file is then sorted on the addresses of tuples of the unsorted relation, allowing efficient retrieval of the corresponding tuples, in physical storage order, to complete the join. Extensions of the technique to handle two unsorted relations are left as an exercise for you.

13.5.5 Hash Join

Like the merge join algorithm, the hash join algorithm can be used to implement natural joins and equi-joins. In the hash join algorithm, a hash function h is used to partition tuples of both relations. The basic idea is to partition the tuples of each of the relations into sets that have the same hash value on the join attributes.

We assume that

- h is a hash function mapping *JoinAttrs* values to $\{0, 1, \ldots, n_h\}$, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.
- $H_{r_0}, H_{r_1}, \ldots, H_{r_{n_h}}$ denote partitions of r tuples, each initially empty. Each tuple $t_r \in r$ is put in partition H_{r_i}, where $i = h(t_r[\mathit{JoinAttrs}])$.
- $H_{s_0}, H_{s_1}, ..., H_{s_{n_h}}$ denote partitions of s tuples, each initially empty. Each tuple $t_s \in s$ is put in partition H_{s_i}, where $i = h(t_s[\mathit{JoinAttrs}])$.

The hash function h should have the "goodness" properties of randomness and uniformity that we discussed in Chapter 12. Figure 13.8 depicts the partitioning of the relations.

The idea behind the hash join algorithm is this: Suppose that an r tuple and an s tuple satisfy the join condition; then, they will have the same value for the join attributes. If that value is hashed to some value i, the r tuple has to be in H_{r_i} and the s tuple in H_{s_i}. Therefore, r tuples in H_{r_i} need only to be compared with s tuples in H_{s_i}; they do not need to be compared with s tuples in any other partition.

For example, if d is a tuple in *depositor*, c a tuple in *customer*, and h a hash function on the *customer-name* attributes of the tuples, then d and c must be tested only if

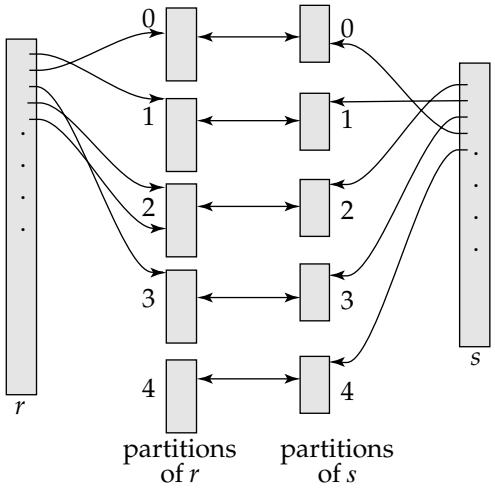


Figure 13.8 Hash partitioning of relations.

$h(c) = h(d)$. If $h(c) \neq h(d)$, then *c* and *d* must have different values for *customer-name*. However, if $h(c) = h(d)$, we must test *c* and *d* to see whether the values in their join attributes are the same, since it is possible that *c* and *d* have different *customer-name*s that have the same hash value.

Figure 13.9 shows the details of the **hash join** algorithm to compute the natural join of relations r and s. As in the merge join algorithm, $t_r \bowtie t_s$ denotes the concatenation of the attributes of tuples t_r and t_s, followed by projecting out repeated attributes. After the partitioning of the relations, the rest of the hash join code performs a separate indexed nested-loop join on each of the partition pairs i, for $i = 0, \ldots, n_h$. To do so, it first **builds** a hash index on each H_{s_i}, and then **probes** (that is, looks up H_{s_i}) with tuples from H_{r_i}. The relation s is the **build input**, and r is the **probe input**.

The hash index on H_{s_i} is built in memory, so there is no need to access the disk to retrieve the tuples. The hash function used to build this hash index is different from the hash function h used earlier, but is still applied to only the join attributes. In the course of the indexed nested-loop join, the system uses this hash index to retrieve records that will match records in the probe input.

The build and probe phases require only a single pass through both the build and probe inputs. It is straightforward to extend the hash join algorithm to compute general equi-joins.

The value n_h must be chosen to be large enough such that, for each i, the tuples in the partition H_{s_i} of the build relation, along with the hash index on the partition, will fit in memory. It is not necessary for the partitions of the probe relation to fit in memory. Clearly, it is best to use the smaller input relation as the build relation. If the size of the build relation is b_s blocks, then, for each of the n_h partitions to be of size less than or equal to M, n_h must be at least $\lceil b_s/M \rceil$. More precisely stated, we have

```
/* Partition s */
for each tuple t_s in s do begin
   i := h(t_s[JoinAttrs]);
   H_{s_i} := H_{s_i} ∪ {t_s};
end
/* Partition r */
for each tuple t_r in r do begin
   i := h(t_r[JoinAttrs]);
   H_{r_i} := H_{r_i} ∪ {t_r};
end
/* Perform join on each partition */
for i := 0 to n_h do begin
   read H_{s_i} and build an in-memory hash index on it
   for each tuple t_r in H_{r_i} do begin
      probe the hash index on H_{s_i} to locate all tuples t_s
         such that t_s[JoinAttrs] = t_r[JoinAttrs]
      for each matching tuple t_s in H_{s_i} do begin
         add t_r ⋈ t_s to the result
      end
   end
end
```

Figure 13.9 Hash join.

to account for the extra space occupied by the hash index on the partition as well, so n_h should be correspondingly larger. For simplicity, we sometimes ignore the space requirement of the hash index in our analysis.

13.5.5.1 Recursive Partitioning

If the value of n_h is greater than or equal to the number of page frames of memory, the relations cannot be partitioned in one pass, since there will not be enough buffer pages. Instead, partitioning has to be done in repeated passes. In one pass, the input can be split into at most as many partitions as there are page frames available for use as output buffers. Each bucket generated by one pass is separately read in and partitioned again in the next pass, to create smaller partitions. The hash function used in a pass is, of course, different from the one used in the previous pass. The system repeats this splitting of the input until each partition of the build input fits in memory. Such partitioning is called **recursive partitioning**.

A relation does not need recursive partitioning if $M > n_h + 1$, or equivalently $M > (b_s/M) + 1$, which simplifies (approximately) to $M > \sqrt{b_s}$. For example, consider a memory size of 12 megabytes, divided into 4-kilobyte blocks; it would contain a total of 3000 blocks. We can use a memory of this size to partition relations of size 9 million blocks, which is 36 gigabytes. Similarly, a relation of size 1 gigabyte requires $\sqrt{250000}$ blocks, or about 2 megabytes, to avoid recursive partitioning.

13.5.5.2 Handling of Overflows

Hash-table overflow occurs in partition i of the build relation s if the hash index on H_{s_i} is larger than main memory. Hash-table overflow can occur if there are many tuples in the build relation with the same values for the join attributes, or if the hash function does not have the properties of randomness and uniformity. In either case, some of the partitions will have more tuples than the average, whereas others will have fewer; partitioning is then said to be **skewed**.

We can handle a small amount of skew by increasing the number of partitions so that the expected size of each partition (including the hash index on the partition) is somewhat less than the size of memory. The number of partitions is therefore increased by a small value called the **fudge factor**, which is usually about 20 percent of the number of hash partitions computed as described in Section 13.5.5.

Even if we are conservative on the sizes of the partitions, by using a fudge factor, overflows can still occur. Hash-table overflows can be handled by either *overflow resolution* or *overflow avoidance*. **Overflow resolution** is performed during the build phase, if a hash-index overflow is detected. Overflow resolution proceeds in this way: If H_{s_i}, for any i, is found to be too large, it is further partitioned into smaller partitions by using a different hash function. Similarly, H_{r_i} is also partitioned using the new hash function, and only tuples in the matching partitions need to be joined.

In contrast, **overflow avoidance** performs the partitioning carefully, so that overflows never occur during the build phase. In overflow avoidance, the build relation s is initially partitioned into many small partitions, and then some partitions are combined in such a way that each combined partition fits in memory. The probe relation r is partitioned in the same way as the combined partitions on s, but the sizes of H_{r_i} do not matter.

If a large number of tuples in s have the same value for the join attributes, the resolution and avoidance techniques may fail on some partitions. In that case, instead of creating an in-memory hash index and using a nested-loop join to join the partitions, we can use other join techniques, such as block nested-loop join, on those partitions.

13.5.5.3 Cost of Hash Join

We now consider the cost of a hash join. Our analysis assumes that there is no hash-table overflow. First, consider the case where recursive partitioning is not required. The partitioning of the two relations r and s calls for a complete reading of both relations, and a subsequent writing back of them. This operation requires $2(b_r + b_s)$ block accesses, where b_r and b_s denote the number of blocks containing records of relations r and s respectively. The build and probe phases read each of the partitions once, calling for a further $b_r + b_s$ accesses. The number of blocks occupied by partitions could be slightly more than $b_r + b_s$, as a result of partially filled blocks. Accessing such partially filled blocks can add an overhead of at most $2n_h$ for each of the relations, since each of the n_h partitions could have a partially filled block that has to be written and read back. Thus, the cost estimate for a hash join is

$$3(b_r + b_s) + 4n_h$$

The overhead $4n_h$ is quite small compared to $b_r + b_s$, and can be ignored.

Now consider the case where recursive partitioning is required. Each pass reduces the size of each of the partitions by an expected factor of $M - 1$; and passes are repeated until each partition is of size at most M blocks. The expected number of passes required for partitioning s is therefore $\lceil \log_{M-1}(b_s) - 1 \rceil$. Since, in each pass, every block of s is read in and written out, the total block transfers for partitioning of s is $2b_s\lceil \log_{M-1}(b_s) - 1 \rceil$. The number of passes for partitioning of r is the same as the number of passes for partitioning of s, therefore the cost estimate for the join is

$$2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$

Consider, for example, the join *customer* $\bowtie$ *depositor*. With a memory size of 20 blocks, *depositor* can be partitioned into five partitions, each of size 20 blocks, which size will fit into memory. Only one pass is required for the partitioning. The relation *customer* is similarly partitioned into five partitions, each of size 80. Ignoring the cost of writing partially filled blocks, the cost is $3(100 + 400) = 1500$ block transfers.

The hash join can be improved if the main memory size is large. When the entire build input can be kept in main memory, n_h can be set to 0; then, the hash join algorithm executes quickly, without partitioning the relations into temporary files, regardless of the probe input's size. The cost estimate goes down to $b_r + b_s$.

13.5.5.4 Hybrid Hash–Join

The **hybrid hash–join** algorithm performs another optimization; it is useful when memory sizes are relatively large, but not all of the build relation fits in memory. The partitioning phase of the hash join algorithm needs one block of memory as a buffer for each partition that is created, and one block of memory as an input buffer. Hence, a total of $n_h + 1$ blocks of memory are needed for the partitioning the two relations. If memory is larger than $n_h + 1$, we can use the rest of memory ($M - n_h - 1$ blocks) to buffer the first partition of the build input (that is, H_{s_0}), so that it will not need to be written out and read back in. Further, the hash function is designed in such a way that the hash index on H_{s_0} fits in $M - n_h - 1$ blocks, in order that, at the end of partitioning of s, H_{s_0} is completely in memory and a hash index can be built on H_{s_0}.

When the system partitions r it again does not write tuples in H_{r_0} to disk; instead, as it generates them, the system uses them to probe the memory-resident hash index on H_{s_0}, and to generate output tuples of the join. After they are used for probing, the tuples can be discarded, so the partition H_{r_0} does not occupy any memory space. Thus, a write and a read access have been saved for each block of both H_{r_0} and H_{s_0}. The system writes out tuples in the other partitions as usual, and joins them later. The savings of hybrid hash–join can be significant if the build input is only slightly bigger than memory.

If the size of the build relation is b_s, n_h is approximately equal to b_s/M. Thus, hybrid hash–join is most useful if $M >> b_s/M$, or $M >> \sqrt{b_s}$, where the notation $>>$ denotes *much larger than*. For example, suppose the block size is 4 kilobytes, and the build relation size is 1 gigabyte. Then, the hybrid hash–join algorithm is useful if the size of memory is significantly more than 2 megabytes; memory sizes of 100 megabytes or more are common on computers today.

Consider the join *customer* $\bowtie$ *depositor* again. With a memory size of 25 blocks, *depositor* can be partitioned into five partitions, each of size 20 blocks, and the first of the partitions of the build relation can be kept in memory. It occupies 20 blocks of memory; one block is for input and one block each is for buffering the other four partitions. The relation *customer* can be similarly partitioned into five partitions each of size 80, the first of which the system uses right away for probing, instead of writing it out and reading it back in. Ignoring the cost of writing partially filled blocks, the cost is $3(80 + 320) + 20 + 80 = 1300$ block transfers, instead of 1500 block transfers without the hybrid hashing optimization.

13.5.6 Complex Joins

Nested-loop and block nested-loop joins can be used regardless of the join conditions. The other join techniques are more efficient than the nested-loop join and its variants, but can handle only simple join conditions, such as natural joins or equijoins. We can implement joins with complex join conditions, such as conjunctions and disjunctions, by using the efficient join techniques, if we apply the techniques developed in Section 13.3.4 for handling complex selections.

Consider the following join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$$

One or more of the join techniques described earlier may be applicable for joins on the individual conditions $r \bowtie_{\theta_1} s$, $r \bowtie_{\theta_2} s$, $r \bowtie_{\theta_3} s$, and so on. We can compute the overall join by first computing the result of one of these simpler joins $r \bowtie_{\theta_i} s$; each pair of tuples in the intermediate result consists of one tuple from r and one from s. The result of the complete join consists of those tuples in the intermediate result that satisfy the remaining conditions

$$\theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \cdots \wedge \theta_n$$

These conditions can be tested as tuples in $r \bowtie_{\theta_i} s$ are being generated.

A join whose condition is disjunctive can be computed in this way: Consider

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s$$

The join can be computed as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \cdots \cup (r \bowtie_{\theta_n} s)$$

Section 13.6 describes algorithms for computing the union of relations.

13.6 Other Operations

Other relational operations and extended relational operations—such as duplicate elimination, projection, set operations, outer join, and aggregation—can be implemented as outlined in Sections 13.6.1 through 13.6.5.

13.6.1 Duplicate Elimination

We can implement duplicate elimination easily by sorting. Identical tuples will appear adjacent to each other during sorting, and all but one copy can be removed. With external sort–merge, duplicates found while a run is being created can be removed before the run is written to disk, thereby reducing the number of block transfers. The remaining duplicates can be eliminated during merging, and the final sorted run will have no duplicates. The worst-case cost estimate for duplicate elimination is the same as the worst-case cost estimate for sorting of the relation.

We can also implement duplicate elimination by hashing, as in the hash join algorithm. First, the relation is partitioned on the basis of a hash function on the whole tuple. Then, each partition is read in, and an in-memory hash index is constructed. While constructing the hash index, a tuple is inserted only if it is not already present. Otherwise, the tuple is discarded. After all tuples in the partition have been processed, the tuples in the hash index are written to the result. The cost estimate is the same as that for the cost of processing (partitioning and reading each partition) of the build relation in a hash join.

Because of the relatively high cost of duplicate elimination, SQL requires an explicit request by the user to remove duplicates; otherwise, the duplicates are retained.

13.6.2 Projection

We can implement projection easily by performing projection on each tuple, which gives a relation that could have duplicate records, and then removing duplicate records. Duplicates can be eliminated by the methods described in Section 13.6.1. If the attributes in the projection list include a key of the relation, no duplicates will exist; hence, duplicate elimination is not required. Generalized projection (which was discussed in Section 3.3.1) can be implemented in the same way as projection.

13.6.3 Set Operations

We can implement the *union*, *intersection*, and *set-difference* operations by first sorting both relations, and then scanning once through each of the sorted relations to produce the result. In $r \cup s$, when a concurrent scan of both relations reveals the same tuple in both files, only one of the tuples is retained. The result of $r \cap s$ will contain only those tuples that appear in both relations. We implement *set difference*, $r - s$, similarly, by retaining tuples in r only if they are absent in s.

For all these operations, only one scan of the two input relations is required, so the cost is $b_r + b_s$. If the relations are not sorted initially, the cost of sorting has to be included. Any sort order can be used in evaluation of set operations, provided that both inputs have that same sort order.

Hashing provides another way to implement these set operations. The first step in each case is to partition the two relations by the same hash function, and thereby create the partitions $H_{r_0}, H_{r_1}, \ldots, H_{r_{n_h}}$ and $H_{s_0}, H_{s_1}, \ldots, H_{s_{n_h}}$. Depending on the operation, the system then takes these steps on each partition $i = 0, 1 \ldots, n_h$:

- $r \cup s$
  1. Build an in-memory hash index on H_{r_i}.
  2. Add the tuples in H_{s_i} to the hash index only if they are not already present.
  3. Add the tuples in the hash index to the result.
- $r \cap s$
  1. Build an in-memory hash index on H_{r_i}.
  2. For each tuple in H_{s_i}, probe the hash index, and output the tuple to the result only if it is already present in the hash index.
- $r - s$
  1. Build an in-memory hash index on H_{r_i}.
  2. For each tuple in H_{s_i}, probe the hash index, and, if the tuple is present in the hash index, delete it from the hash index.
  3. Add the tuples remaining in the hash index to the result.

13.6.4 Outer Join

Recall the *outer-join operations* described in Section 3.3.3. For example, the natural left outer join *customer* $\supset\!\bowtie$ *depositor* contains the join of *customer* and *depositor*, and, in addition, for each *customer* tuple t that has no matching tuple in *depositor* (that is, where *customer-name* is not in *depositor*), the following tuple t_1 is added to the result. For all attributes in the schema of *customer*, tuple t_1 has the same values as tuple t. The remaining attributes (from the schema of *depositor*) of tuple t_1 contain the value null.

We can implement the outer-join operations by using one of two strategies:

1. Compute the corresponding join, and then add further tuples to the join result to get the outer-join result. Consider the left outer-join operation and two relations: $r(R)$ and $s(S)$. To evaluate $r \supset\!\bowtie_\theta s$, we first compute $r \bowtie_\theta s$, and save that result as temporary relation q_1. Next, we compute $r - \Pi_R(q_1)$, which gives tuples in r that did not participate in the join. We can use any of the algorithms for computing the joins, projection, and set difference described earlier to compute the outer joins. We pad each of these tuples with null values for attributes from s, and add it to q_1 to get the result of the outer join.

   The right outer-join operation $r \bowtie\!\subset_\theta s$ is equivalent to $s \supset\!\bowtie_\theta r$, and can therefore be implemented in a symmetric fashion to the left outer join. We can implement the full outer-join operation $r \supset\!\bowtie\!\subset_\theta s$ by computing the join $r \bowtie s$, and then adding the extra tuples of both the left and right outer-join operations, as before.

2. Modify the join algorithms. It is easy to extend the nested-loop join algorithms to compute the left outer join: Tuples in the outer relation that do not match any tuple in the inner relation are written to the output after being padded with null values. However, it is hard to extend the nested-loop join to compute the full outer join.

Natural outer joins and outer joins with an equi-join condition can be computed by extensions of the merge join and hash join algorithms. Merge join can be extended to compute the full outer join as follows: When the merge of the two relations is being done, tuples in either relation that did not match any tuple in the other relation can be padded with nulls and written to the output. Similarly, we can extend merge join to compute the left and right outer joins by writing out nonmatching tuples (padded with nulls) from only one of the relations. Since the relations are sorted, it is easy to detect whether or not a tuple matches any tuples from the other relation. For example, when a merge join of *customer* and *depositor* is done, the tuples are read in sorted order of *customer-name*, and it is easy to check, for each tuple, whether there is a matching tuple in the other.

The cost estimates for implementing outer joins using the merge join algorithm are the same as are those for the corresponding join. The only difference lies in size of the result, and therefore in the block transfers for writing it out, which we did not count in our earlier cost estimates.

The extension of the hash join algorithm to compute outer joins is left for you to do as an exercise (Exercise 13.11).

13.6.5 Aggregation

Recall the aggregation operator $\mathcal{G}$, discussed in Section 3.3.2. For example, the operation

$$_{branch\text{-}name}\mathcal{G}_{\mathbf{sum}(balance)}(account)$$

groups *account* tuples by branch, and computes the total balance of all the accounts at each branch.

The aggregation operation can be implemented in the same way as duplicate elimination. We use either sorting or hashing, just as we did for duplicate elimination, but based on the grouping attributes (*branch-name* in the preceding example). However, instead of eliminating tuples with the same value for the grouping attribute, we gather them into groups, and apply the aggregation operations on each group to get the result.

The cost estimate for implementing the aggregation operation is the same as the cost of duplicate elimination, for aggregate functions such as **min**, **max**, **sum**, **count**, and **avg**.

Instead of gathering all the tuples in a group and then applying the aggregation operations, we can implement the aggregation operations **sum**, **min**, **max**, **count**, and **avg** on the fly as the groups are being constructed. For the case of **sum**, **min**, and **max**, when two tuples in the same group are found, the system replaces them by a single tuple containing the **sum**, **min**, or **max**, respectively, of the columns being aggregated. For the **count** operation, it maintains a running count for each group for which a tuple has been found. Finally, we implement the **avg** operation by computing the sum and the count values on the fly, and finally dividing the sum by the count to get the average.

If all tuples of the result will fit in memory, both the sort-based and the hash-based implementations do not need to write any tuples to disk. As the tuples are read in, they can be inserted in a sorted tree structure or in a hash index. When we use on the fly aggregation techniques, only one tuple needs to be stored for each of the groups. Hence, the sorted tree structure or hash index will fit in memory, and the aggregation can be processed with just b_r block transfers, instead of with the $3b_r$ transfers that would be required otherwise.

13.7 Evaluation of Expressions

So far, we have studied how individual relational operations are carried out. Now we consider how to evaluate an expression containing multiple operations. The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use. A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk. An alternative approach is to evaluate several operations simultaneously in a **pipeline**, with the results of one operation passed on to the next, without the need to store a temporary relation.

In Sections 13.7.1 and 13.7.2, we consider both the *materialization* approach and the *pipelining* approach. We shall see that the costs of these approaches can differ substantially, but also that there are cases where only the materialization approach is feasible.

13.7.1 Materialization

It is easiest to understand intuitively how to evaluate an expression by looking at a pictorial representation of the expression in an **operator tree**. Consider the expression

$$\Pi_{customer\text{-}name}\ (\sigma_{balance<2500}\ (account) \bowtie\ customer)$$

in Figure 13.10.

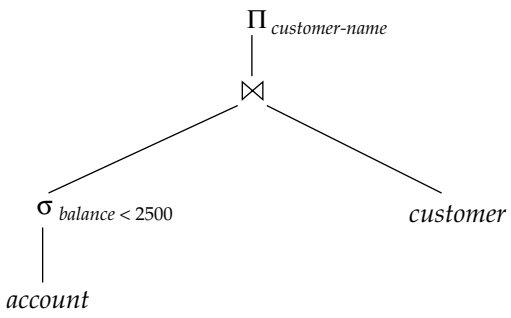


Figure 13.10 Pictorial representation of an expression.

If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree). In our example, there is only one such operation; the selection operation on *account*. The inputs to the lowest-level operations are relations in the database. We execute these operations by the algorithms that we studied earlier, and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. In our example, the inputs to the join are the *customer* relation and the temporary relation created by the selection on *account*. The join can now be evaluated, creating another temporary relation.

By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression. In our example, we get the final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.

Evaluation as just described is called **materialized evaluation**, since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.

The cost of a materialized evaluation is not simply the sum of the costs of the operations involved. When we computed the cost estimates of algorithms, we ignored the cost of writing the result of the operation to disk. To compute the cost of evaluating an expression as done here, we have to add the costs of all the operations, as well as the cost of writing the intermediate results to disk. We assume that the records of the result accumulate in a buffer, and, when the buffer is full, they are written to disk. The cost of writing out the result can be estimated as n_r/f_r, where n_r is the estimated number of tuples in the result relation r, and f_r is the *blocking factor* of the result relation, that is, the number of records of r that will fit in a block.

Double buffering (using two buffers, with one continuing execution of the algorithm while the other is being written out) allows the algorithm to execute more quickly by performing CPU activity in parallel with I/O activity.

13.7.2 Pipelining

We can improve query-evaluation efficiency by reducing the number of temporary files that are produced. We achieve this reduction by combining several relational operations into a *pipeline* of operations, in which the results of one operation are passed along to the next operation in the pipeline. Evaluation as just described is called **pipelined evaluation**. Combining operations into a pipeline eliminates the cost of reading and writing temporary relations.

For example, consider the expression $(\Pi_{a1,a2}(r \bowtie s))$. If materialization were applied, evaluation would involve creating a temporary relation to hold the result of the join, and then reading back in the result to perform the projection. These operations can be combined: When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, we avoid creating the intermediate result, and instead create the final result directly.

13.7.2.1 Implementation of Pipelining

We can implement a pipeline by constructing a single, complex operation that combines the operations that constitute the pipeline. Although this approach may be feasible for various frequently occurring situations, it is desirable in general to reuse the code for individual operations in the construction of a pipeline. Therefore, each operation in the pipeline is modeled as a separate process or thread within the system, which takes a stream of tuples from its pipelined inputs, and generates a stream of tuples for its output. For each pair of adjacent operations in the pipeline, the system creates a buffer to hold tuples being passed from one operation to the next.

In the example of Figure 13.10, all three operations can be placed in a pipeline, which passes the results of the selection to the join as they are generated. In turn, it passes the results of the join to the projection as they are generated. The memory requirements are low, since results of an operation are not stored for long. However, as a result of pipelining, the inputs to the operations are not available all at once for processing.

Pipelines can be executed in either of two ways:

1. Demand driven
2. Producer driven

In a **demand-driven pipeline**, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple. If the inputs of the operation are not pipelined, the next tuple(s) to be returned can be computed from the input relations, while the system keeps track of what has been returned so far. If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs. Using the tuples received from its pipelined inputs, the operation computes tuples for its output, and passes them up to its parent.

In a **producer-driven pipeline**, operations do not wait for requests to produce tuples, but instead generate the tuples **eagerly**. Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer, until the buffer is full. An operation at any other level of a pipeline generates output tuples when it gets input tuples from lower down in the pipeline, until its output buffer is full. Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer. In either case, once the output buffer is full, the operation waits until its parent operation removes tuples from the buffer, so that the buffer has space for more tuples. At this point, the operation generates more tuples, until the buffer is full again. The operation repeats this process until all the output tuples have been generated.

It is necessary for the system to switch between operations only when an output buffer is full, or an input buffer is empty and more input tuples are needed to generate any more output tuples. In a parallel-processing system, operations in a pipeline may be run concurrently on distinct processors (see Chapter 20).

Using producer-driven pipelining can be thought of as **pushing** data up an operation tree from below, whereas using demand-driven pipelining can be thought of as

pulling data up an operation tree from the top. Whereas tuples are generated *eagerly* in producer-driven pipelining, they are generated **lazily**, on demand, in demand-driven pipelining.

Each operation in a demand-driven pipeline can be implemented as an **iterator**, which provides the following functions: open(), next(), and close(). After a call to open(), each call to next() returns the next output tuple of the operation. The implementation of the operation in turn calls open() and next() on its inputs, to get its input tuples when required. The function close() tells an iterator that no more tuples are required. The iterator maintains the **state** of its execution in between calls, so that successive next() requests receive successive result tuples.

For example, for an iterator implementing the select operation using linear search, the open() operation starts a file scan, and the iterator's state records the point to which the file has been scanned. When the next() function is called, the file scan continues from after the previous point; when the next tuple satisfying the selection is found by scanning the file, the tuple is returned after storing the point where it was found in the iterator state. A merge–join iterator's open() operation would open its inputs, and if they are not already sorted, it would also sort the inputs. On calls to next(), it would return the next pair of matching tuples. The state information would consist of up to where each input had been scanned.

Details of the implementation of iterators are left for you to complete in Exercise 13.12. Demand-driven pipelining is used more commonly than producer-driven pipelining, because it is easier to implement.

13.7.2.2 Evaluation Algorithms for Pipelining

Consider a join operation whose left-hand–side input is pipelined. Since it is pipelined, the input is not available all at once for processing by the join operation. This unavailability limits the choice of join algorithm to be used. Merge join, for example, cannot be used if the inputs are not sorted, since it is not possible to sort a relation until all the tuples are available—thus, in effect, turning pipelining into materialization. However, indexed nested-loop join can be used: As tuples are received for the left-hand side of the join, they can be used to index the right-hand–side relation, and to generate tuples in the join result. This example illustrates that choices regarding the algorithm used for an operation and choices regarding pipelining are not independent.

The restrictions on the evaluation algorithms that are eligible for use are a limiting factor for pipelining. As a result, despite the apparent advantages of pipelining, there are cases where materialization achieves lower overall cost. Suppose that the join of r and s is required, and input r is pipelined. If indexed nested-loop join is used to support pipelining, one access to disk may be needed for every tuple in the pipelined input relation. The cost of this technique is $n_r * \mathrm{HT}_i$, where HT_i is the height of the index on s. With materialization, the cost of writing out r would be b_r. With a join technique such as hash join, it may be possible to perform the join with a cost of about $3(b_r + b_s)$. If n_r is substantially more than $4b_r + 3b_s$, materialization would be cheaper.

```
done_r := false;
done_s := false;
r := ∅;
s := ∅;
result := ∅;
while not done_r or not done_s do
    begin
        if queue is empty, then wait until queue is not empty;
        t := top entry in queue;
        if t = End_r then done_r := true
          else if t = End_s then done_s := true
              else if t is from input r
                  then
                      begin
                          r := r ∪ {t};
                          result := result ∪ ({t} ⋈ s);
                      end
                  else /* t is from input s */
                      begin
                          s := s ∪ {t};
                          result := result ∪ (r ⋈ {t});
                      end
    end
```

Figure 13.11 Pipelined join algorithm.

The effective use of pipelining requires the use of evaluation algorithms that can generate output tuples even as tuples are received for the inputs to the operation. We can distinguish between two cases:

1. Only one of the inputs to a join is pipelined.
2. Both inputs to the join are pipelined.

If only one of the inputs to a join is pipelined, indexed nested-loop join is a natural choice. If the pipelined input tuples are sorted on the join attributes, and the join condition is an equi-join, merge join can also be used. Hybrid hash–join can be used too, with the pipelined input as the probe relation. However, tuples that are not in the first partition will be output only after the entire pipelined input relation is received. Hybrid hash–join is useful if the nonpipelined input fits entirely in memory, or if at least most of that input fits in memory.

If both inputs are pipelined, the choice of join algorithms is more restricted. If both inputs are sorted on the join attribute, and the join condition is an equi-join, merge join can be used. Another alternative is the **pipelined join** technique, shown in Figure 13.11. The algorithm assumes that the input tuples for both input relations, r and s, are pipelined. Tuples made available for both relations are queued for processing in a single queue. Special queue entries, called End_r and End_s, which serve as end-of-file

markers, are inserted in the queue after all tuples from r and s (respectively) have been generated. For efficient evaluation, appropriate indices should be built on the relations r and s. As tuples are added to r and s, the indices must be kept up to date.

13.8 Summary

- The first action that the system must perform on a query is to translate the query into its internal form, which (for relational database systems) is usually based on the relational algebra. In the process of generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of relations in the database, and so on. If the query was expressed in terms of a view, the parser replaces all references to the view name with the relational-algebra expression to compute the view.
- Given a query, there are generally a variety of methods for computing the answer. It is the responsibility of the query optimizer to transform the query as entered by the user into an equivalent query that can be computed more efficiently. Chapter 14 covers query optimization.
- We can process simple selection operations by performing a linear scan, by doing a binary search, or by making use of indices. We can handle complex selections by computing unions and intersections of the results of simple selections.
- We can sort relations larger than memory by the external merge–sort algorithm.
- Queries involving a natural join may be processed in several ways, depending on the availability of indices and the form of physical storage for the relations.
  - If the join result is almost as large as the Cartesian product of the two relations, a *block nested-loop* join strategy may be advantageous.
  - If indices are available, the *indexed nested-loop* join can be used.
  - If the relations are sorted, a *merge join* may be desirable. It may be advantageous to sort a relation prior to join computation (so as to allow use of the merge join strategy).
  - The *hash join* algorithm partitions the relations into several pieces, such that each piece of one of the relations fits in memory. The partitioning is carried out with a hash function on the join attributes, so that corresponding pairs of partitions can be joined independently.
- Duplicate elimination, projection, set operations (union, intersection and difference), and aggregation can be done by sorting or by hashing.
- Outer join operations can be implemented by simple extensions of join algorithms.
- Hashing and sorting are dual, in the sense that any operation such as duplicate elimination, projection, aggregation, join, and outer join that can be

implemented by hashing can also be implemented by sorting, and vice versa; that is, any operation that can be implemented by sorting can also be implemented by hashing.

- An expression can be evaluated by means of materialization, where the system computes the result of each subexpression and stores it on disk, and then uses it to compute the result of the parent expression.
- Pipelining helps to avoid writing the results of many subexpressions to disk, by using the results in the parent expression even as they are being generated.

Review Terms

- Query processing
- Evaluation primitive
- Query-execution plan
- Query-evaluation plan
- Query-execution engine
- Measures of query cost
- Sequential I/O
- Random I/O
- File scan
- Linear search
- Binary search
- Selections using indices
- Access paths
- Index scans
- Conjunctive selection
- Disjunctive selection
- Composite index
- Intersection of identifiers
- External sorting
- External sort–merge
- Runs
- N-way merge
- Equi-join
- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge join
- Sort–merge join
- Hybrid merge–join
- Hash join
  - Build
  - Probe
  - Build input
  - Probe input
  - Recursive partitioning
  - Hash-table overflow
  - Skew
  - Fudge factor
  - Overflow resolution
  - Overflow avoidance
- Hybrid hash–join
- Operator tree
- Materialized evaluation
- Double buffering
- Pipelined evaluation
  - Demand-driven pipeline (lazy, pulling)
  - Producer-driven pipeline (eager, pushing)
  - Iterator
- Pipelined join

Exercises

13.1 Why is it not desirable to force users to make an explicit choice of a query-processing strategy? Are there cases in which it *is* desirable for users to be aware of the costs of competing query-processing strategies? Explain your answer.

13.2 Consider the following SQL query for our bank database:

select *T.branch-name*
from *branch T*, *branch S*
where *T.assets* $>$ *S.assets* **and** *S.branch-city* = "Brooklyn"

Write an efficient relational-algebra expression that is equivalent to this query. Justify your choice.

13.3 What are the advantages and disadvantages of hash indices relative to B^+-tree indices? How might the type of index available influence the choice of a query-processing strategy?

13.4 Assume (for simplicity in this exercise) that only one tuple fits in a block and memory holds at most 3 page frames. Show the runs created on each pass of the sort-merge algorithm, when applied to sort the following tuples on the first attribute: (kangaroo, 17), (wallaby, 21), (emu, 1), (wombat, 13), (platypus, 3), (lion, 8), (warthog, 4), (zebra, 11), (meerkat, 6), (hyena, 9), (hornbill, 2), (baboon, 12).

13.5 Let relations $r_1(A, B, C)$ and $r_2(C, D, E)$ have the following properties: r_1 has 20,000 tuples, r_2 has 45,000 tuples, 25 tuples of r_1 fit on one block, and 30 tuples of r_2 fit on one block. Estimate the number of block accesses required, using each of the following join strategies for $r_1 \bowtie r_2$:

a. Nested-loop join
b. Block nested-loop join
c. Merge join
d. Hash join

13.6 Design a variant of the hybrid merge–join algorithm for the case where both relations are not physically sorted, but both have a sorted secondary index on the join attributes.

13.7 The indexed nested-loop join algorithm described in Section 13.5.3 can be inefficient if the index is a secondary index, and there are multiple tuples with the same value for the join attributes. Why is it inefficient? Describe a way, using sorting, to reduce the cost of retrieving tuples of the inner relation. Under what conditions would this algorithm be more efficient than hybrid merge–join?

13.8 Estimate the number of block accesses required by your solution to Exercise 13.6 for $r_1 \bowtie r_2$, where r_1 and r_2 are as defined in Exercise 13.5.

13.9 Let r and s be relations with no indices, and assume that the relations are not sorted. Assuming infinite memory, what is the lowest cost way (in terms of I/O operations) to compute $r \bowtie s$? What is the amount of memory required for this algorithm?

13.10 Suppose that a B^+-tree index on *branch-city* is available on relation *branch*, and that no other index is available. List different ways to handle the following selections that involve negation?

a. $\sigma_{\neg(branch\text{-}city < \text{“Brooklyn”})}(branch)$
b. $\sigma_{\neg(branch\text{-}city = \text{“Brooklyn”})}(branch)$
c. $\sigma_{\neg(branch\text{-}city < \text{“Brooklyn”} \vee assets < 5000)}(branch)$

13.11 The hash join algorithm as described in Section 13.5.5 computes the natural join of two relations. Describe how to extend the hash join algorithm to compute the natural left outer join, the natural right outer join and the natural full outer join. (Hint: Keep extra information with each tuple in the hash index, to detect whether any tuple in the probe relation matches the tuple in the hash index.) Try out your algorithm on the *customer* and *depositor* relations.

13.12 Write pseudocode for an iterator that implements indexed nested-loop join, where the outer relation is pipelined. Use the standard iterator functions in your pseudocode. Show what state information the iterator must maintain between calls.

13.13 Design sorting based and hashing algorithms for computing the division operation.

Bibliographical Notes

A query processor must parse statements in the query language, and must translate them into an internal form. Parsing of query languages differs little from parsing of traditional programming languages. Most compiler texts, such as Aho et al. [1986], cover the main parsing techniques, and present optimization from a programming-language point of view.

Knuth [1973] presents an excellent description of external sorting algorithms, including an optimization that can create initial runs that are (on the average) twice the size of memory. Based on performance studies conducted in the mid-1970s, database systems of that period used only nested-loop join and merge join. These studies, which were related to the development of System R, determined that either the nested-loop join or merge join nearly always provided the optimal join method (Blasgen and Eswaran [1976]); hence, these two were the only join algorithms implemented in System R. The System R study, however, did not include an analysis of hash join algorithms. Today, hash joins are considered to be highly efficient.

Hash join algorithms were initially developed for parallel database systems. Hash join techniques are described in Kitsuregawa et al. [1983], and extensions including hybrid hash join are described in Shapiro [1986]. Zeller and Gray [1990] and Davison and Graefe [1994] describe hash join techniques that can adapt to the available mem-

ory, which is important in systems where multiple queries may be running at the same time. Graefe et al. [1998] describes the use of hash joins and *hash teams*, which allow pipelining of hash-joins by using the same partitioning for all hash-joins in a pipeline sequence, in the Microsoft SQL Server.

Graefe [1993] presents an excellent survey of query-evaluation techniques. An earlier survey of query-processing techniques appears in Jarke and Koch [1984].

Query processing in main memory database is covered by DeWitt et al. [1984] and Whang and Krishnamurthy [1990]. Kim [1982] and Kim [1984] describe join strategies and the optimal use of available main memory.

CHAPTER 14

Query Optimization

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex. We do not expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query-evaluation plan that minimizes the cost of query evaluation. This is where query optimization comes into play.

One aspect of optimization occurs at the relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute. Another aspect is selecting a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation, choosing the specific indices to use, and so on.

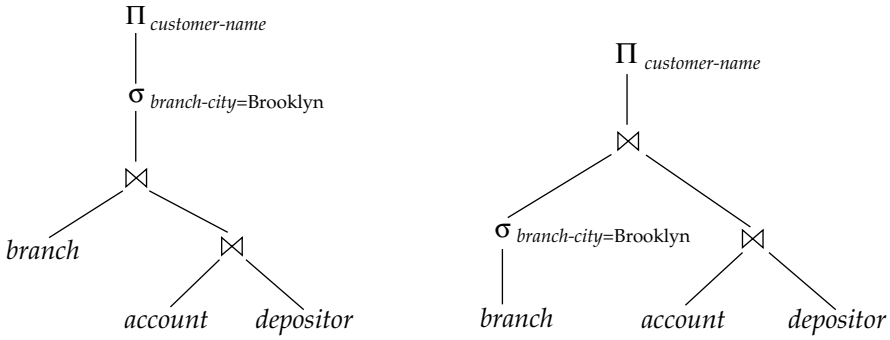
The difference in cost (in terms of evaluation time) between a good strategy and a bad strategy is often substantial, and may be several orders of magnitude. Hence, it is worthwhile for the system to spend a substantial amount of time on the selection of a good strategy for processing a query, even if the query is executed only once.

14.1 Overview

Consider the relational-algebra expression for the query "Find the names of all customers who have an account at any branch located in Brooklyn."

$$\Pi_{customer\text{-}name}\ (\sigma_{branch-city\ =\ \text{"Brooklyn"}}\ (branch \bowtie\ (account \bowtie\ depositor)))$$

This expression constructs a large intermediate relation, $branch \bowtie account \bowtie depositor$. However, we are interested in only a few tuples of this relation (those pertaining to branches located in Brooklyn), and in only one of the six attributes of this relation. Since we are concerned with only those tuples in the *branch* relation that pertain to branches located in Brooklyn, we do not need to consider those tuples that do not



(a) Initial expression tree (b) Transformed expression tree

Figure 14.1 Equivalent expressions.

have *branch-city* = "Brooklyn". By reducing the number of tuples of the *branch* relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression

$$\Pi_{customer\text{-}name}\ (\ (\sigma_{branch\text{-}city\ =\ \text{"Brooklyn"}}\ (branch))\ \bowtie\ (account\ \bowtie\ depositor))$$

which is equivalent to our original algebra expression, but which generates smaller intermediate relations. Figure 14.1 depicts the initial and transformed expressions.

Given a relational-algebra expression, it is the job of the query optimizer to come up with a query-evaluation plan that computes the same result as the given expression, and is the least costly way of generating the result (or, at least, is not much costlier than the least costly way).

To choose among different query-evaluation plans, the optimizer has to **estimate** the cost of each evaluation plan. Computing the precise cost of evaluation of a plan is usually not possible without actually evaluating the plan. Instead, optimizers make use of statistical information about the relations, such as relation sizes and index depths, to make a good estimate of the cost of a plan. Disk access, which is slow compared to memory access, usually dominates the cost of processing a query.

In Section 14.2 we describe how to estimate statistics of the results of each operation in a query plan. Using these statistics with the cost formulae in Chapter 13 allows us to estimate the costs of individual operation. The individual costs are combined to determine the estimated cost of evaluating a given relational-algebra expression, as outlined earlier in Section 13.7.

To find the least-costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression, and to choose the least costly one. Generation of query-evaluation plans involves two steps: (1) generating expressions that are logically equivalent to the given expression and (2) annotating the resultant expressions in alternative ways to generate alternative query evaluation plans. The two steps are interleaved in the query optimizer—some expressions are generated and annotated, then further expressions are generated and annotated, and so on.

To implement the first step, the query optimizer must generate expressions equivalent to a given expression. It does so by means of *equivalence rules* that specify how to transform an expression into a logically equivalent one. We describe these rules in Section 14.3.1. In Section 14.4, we describe how to choose a query-evaluation plan. We can choose one based on the estimated cost of the plans. Since the cost is an estimate, the selected plan is not necessarily the least costly plan; however, as long as the estimates are good, the plan is likely to be the least costly one, or not much more costly than it. Such optimization, called **cost-based optimization**, is described in Section 14.4.2.

Materialized views help to speed up processing of certain queries. In Section 14.5, we study how to "maintain" materialized views—that is, to keep them up-to-date—and how to perform query optimization with materialized views.

14.2 Estimating Statistics of Expression Results

The cost of an operation depends on the size and other statistics of its inputs. Given an expression such as $a \bowtie (b \bowtie c)$ to estimate the cost of joining a with $(b \bowtie c)$, we need to have estimates of statistics such as the size of $b \bowtie c$.

In this section we first list some statistics about database relations that are stored in database system catalogs, and then show how to use the statistics to estimate statistics on the results of various relational operations.

One thing that will become clear later in this section is that the estimates are not very accurate, since they are based on assumptions that may not hold exactly. A query evaluation plan that has the lowest estimated execution cost may therefore not actually have the lowest actual execution cost. However, real-world experience has shown that even if estimates are not precise, the plans with the lowest estimated costs usually have actual execution costs that are either the lowest actual execution costs, or are close to the lowest actual execution costs.

14.2.1 Catalog Information

The DBMS catalog stores the following statistical information about database relations:

- n_r, the number of tuples in the relation r.
- b_r, the number of blocks containing tuples of relation r.
- l_r, the size of a tuple of relation r in bytes.
- f_r, the blocking factor of relation r—that is, the number of tuples of relation r that fit into one block.
- $V(A, r)$, the number of distinct values that appear in the relation r for attribute A. This value is the same as the size of $\Pi_A(r)$. If A is a key for relation r, $V(A, r)$ is n_r.

The last statistic, $V(A, r)$, can also be maintained for sets of attributes, if desired, instead of just for individual attributes. Thus, given a set of attributes, $\mathcal{A}$, $V(\mathcal{A}, r)$ is the size of $\Pi_{\mathcal{A}}(r)$.

If we assume that the tuples of relation r are stored together physically in a file, the following equation holds:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Statistics about indices, such as the heights of B^+-tree indices and number of leaf pages in the indices, are also maintained in the catalog.

If we wish to maintain accurate statistics, then, every time a relation is modified, we must also update the statistics. This update incurs a substantial amount of overhead. Therefore, most systems do not update the statistics on every modification. Instead, they update the statistics during periods of light system load. As a result, the statistics used for choosing a query-processing strategy may not be completely accurate. However, if not too many updates occur in the intervals between the updates of the statistics, the statistics will be sufficiently accurate to provide a good estimation of the relative costs of the different plans.

The statistical information noted here is simplified. Real-world optimizers often maintain further statistical information to improve the accuracy of their cost estimates of evaluation plans. For instance, some databases store the distribution of values for each attribute as a **histogram**: in a histogram the values for the attribute are divided into a number of ranges, and with each range the histogram associates the number of tuples whose attribute value lies in that range. As an example of a histogram, the range of values for an attribute *age* of a relation *person* could be divided into $0–9$, $10–19$, $\ldots$, $90–99$ (assuming a maximum age of 99). With each range we store a count of the number of *person* tuples whose *age* values lie in that range. Without such histogram information, an optimizer would have to assume that the distribution of values is uniform; that is, each range has the same count.

14.2.2 Selection Size Estimation

The size estimate of the result of a selection operation depends on the selection predicate. We first consider a single equality predicate, then a single comparison predicate, and finally combinations of predicates.

- $\sigma_{A=a}(r)$: If we assume uniform distribution of values (that is, each value appears with equal probability), the selection result can be estimated to have $n_r/V(A, r)$ tuples, assuming that the value a appears in attribute A of some record of r. The assumption that the value a in the selection appears in some record is generally true, and cost estimates often make it implicitly. However, it is often not realistic to assume that each value appears with equal probability. The *branch-name* attribute in the *account* relation is an example where the assumption is not valid. There is one tuple in the *account* relation for each account. It is reasonable to expect that the large branches have more accounts than smaller branches. Therefore, certain *branch-name* values appear with greater probability than do others. Despite the fact that the uniform-

distribution assumption is often not correct, it is a reasonable approximation of reality in many cases, and it helps us to keep our presentation relatively simple.

- $\sigma_{A \leq v}(r)$: Consider a selection of the form $\sigma_{A \leq v}(r)$. If the actual value used in the comparison (v) is available at the time of cost estimation, a more accurate estimate can be made. The lowest and highest values ($\min(A, r)$ and $\max(A, r)$) for the attribute can be stored in the catalog. Assuming that values are uniformly distributed, we can estimate the number of records that will satisfy the condition $A \leq v$ as 0 if $v < \min(A, r)$, as n_r if $v \geq \max(A, r)$, and

$$n_r \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

otherwise.

In some cases, such as when the query is part of a stored procedure, the value v may not be available when the query is optimized. In such cases, we will assume that approximately one-half the records will satisfy the comparison condition. That is, we assume the result has $n_r/2$ tuples; the estimate may be very inaccurate, but is the best we can do without any further information.

- Complex selections:
  - **Conjunction:** A *conjunctive selection* is a selection of the form

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

We can estimate the result size of such a selection: For each θ_i, we estimate the size of the selection $\sigma_{\theta_i}(r)$, denoted by s_i, as described previously. Thus, the probability that a tuple in the relation satisfies selection condition θ_i is s_i/n_r.

The preceding probability is called the **selectivity** of the selection $\sigma_{\theta_i}(r)$. Assuming that the conditions are *independent* of each other, the probability that a tuple satisfies all the conditions is simply the product of all these probabilities. Thus, we estimate the number of tuples in the full selection as

$$n_r * \frac{s_1 * s_2 * \cdots * s_n}{n_r^n}$$

  - **Disjunction:** A *disjunctive selection* is a selection of the form

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θ_i.

As before, let s_i/n_r denote the probability that a tuple satisfies condition θ_i. The probability that the tuple will satisfy the disjunction is then 1 minus the probability that it will satisfy *none* of the conditions:

$$1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \cdots * (1 - \frac{s_n}{n_r})$$

Multiplying this value by n_r gives us the estimated number of tuples that satisfy the selection.

- **Negation:** In the absence of nulls, the result of a selection $\sigma_{\neg\theta}(r)$ is simply the tuples of r that are not in $\sigma_\theta(r)$. We already know how to estimate the number of tuples in $\sigma_\theta(r)$. The number of tuples in $\sigma_{\neg\theta}(r)$ is therefore estimated to be $n(r)$ minus the estimated number of tuples in $\sigma_\theta(r)$.

  We can account for nulls by estimating the number of tuples for which the condition θ would evaluate to *unknown*, and subtracting that number from the above estimate ignoring nulls. Estimating that number would require extra statistics to be maintained in the catalog.

14.2.3 Join Size Estimation

In this section, we see how to estimate the size of the result of a join.

The Cartesian product $r \times s$ contains $n_r * n_s$ tuples. Each tuple of $r \times s$ occupies $l_r + l_s$ bytes, from which we can calculate the size of the Cartesian product.

Estimating the size of a natural join is somewhat more complicated than estimating the size of a selection or of a Cartesian product. Let $r(R)$ and $s(S)$ be relations.

- If $R \cap S = \emptyset$—that is, the relations have no attribute in common—then $r \bowtie s$ is the same as $r \times s$, and we can use our estimation technique for Cartesian products.
- If $R \cap S$ is a key for R, then we know that a tuple of s will join with at most one tuple from r. Therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s. The case where $R \cap S$ is a key for S is symmetric to the case just described. If $R \cap S$ forms a foreign key of S, referencing R, the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
- The most difficult case is when $R \cap S$ is a key for neither R nor S. In this case, we assume, as we did for selections, that each value appears with equal probability. Consider a tuple t of r, and assume $R \cap S = \{\text{A}\}$. We estimate that tuple t produces

  $$\frac{n_s}{V(A, s)}$$

  tuples in $r \bowtie s$, since this number is the average number of tuples in s with a given value for the attributes A. Considering all the tuples in r, we estimate that there are

  $$\frac{n_r * n_s}{V(A, s)}$$

  tuples in $r \bowtie s$. Observe that, if we reverse the roles of r and s in the preceding estimate, we obtain an estimate of

  $$\frac{n_r * n_s}{V(A, r)}$$

  tuples in $r \bowtie s$. These two estimates differ if $V(A, r) \neq V(A, s)$. If this situation occurs, there are likely to be dangling tuples that do not participate in the join. Thus, the lower of the two estimates is probably the more accurate one.

  The preceding estimate of join size may be too high if the $V(A, r)$ values for attribute A in r have few values in common with the $V(A, s)$ values for

attribute A in s. However, this situation is unlikely to happen in the real world, since dangling tuples either do not exist, or constitute only a small fraction of the tuples, in most real-world relations. More important, the preceding estimate depends on the assumption that each value appears with equal probability. More sophisticated techniques for size estimation have to be used if this assumption does not hold.

We can estimate the size of a theta join $r \bowtie_\theta s$ by rewriting the join as $\sigma_\theta(r \times s)$, and using the size estimates for Cartesian products along with the size estimates for selections, which we saw in Section 14.2.2.

To illustrate all these ways of estimating join sizes, consider the expression

$$depositor \bowtie customer$$

Assume the following catalog information about the two relations:

- $n_{customer} = 10000$.
- $f_{customer} = 25$, which implies that $b_{customer} = 10000/25 = 400$.
- $n_{depositor} = 5000$.
- $f_{depositor} = 50$, which implies that $b_{depositor} = 5000/50 = 100$.
- V(customer-name, depositor) $= 2500$, which implies that, on average, each customer has two accounts.

Also assume that *customer-name* in *depositor* is a foreign key on *customer*.

In our example of *depositor* $\bowtie$ *customer*, *customer-name* in *depositor* is a foreign key referencing *customer*; hence, the size of the result is exactly $n_{depositor}$, which is 5000.

Let us now compute the size estimates for *depositor* $\bowtie$ *customer* without using information about foreign keys. Since V(*customer-name, depositor*) $= 2500$ and V(*customer-name, customer*) $= 10000$, the two estimates we get are $5000 * 10000/2500 = 20,000$ and $5000 * 10000/10000 = 5000$, and we choose the lower one. In this case, the lower of these estimates is the same as that which we computed earlier from information about foreign keys.

14.2.4 Size Estimation for Other Operations

We outline below how to estimate the sizes of the results of other relational algebra operations.

Projection: The estimated size (number of records or number of tuples) of a projection of the form $\Pi_A(r)$ is $V(A, r)$, since projection eliminates duplicates.

Aggregation: The size of ${}_A\mathcal{G}_F(r)$ is simply $V(A, r)$, since there is one tuple in ${}_A\mathcal{G}_F(r)$ for each distinct value of A.

Set operations: If the two inputs to a set operation are selections on the same relation, we can rewrite the set operation as disjunctions, conjunctions, or negations. For example, $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ can be rewritten as $\sigma_{\theta_1 \vee \theta_2}(r)$. Similarly, we

can rewrite intersections as conjunctions, and we can rewrite set difference by using negation, so long as the two relations participating in the set operations are selections on the same relation. We can then use the estimates for selections involving conjunctions, disjunctions, and negation in Section 14.2.2.

If the inputs are not selections on the same relation, we estimate the sizes this way: The estimated size of $r \cup s$ is the sum of the sizes of r and s. The estimated size of $r \cap s$ is the minimum of the sizes of r and s. The estimated size of $r - s$ is the same size as r. All three estimates may be inaccurate, but provide upper bounds on the sizes.

Outer join: The estimated size of $r \supset\!\bowtie s$ is the size of $r \bowtie s$ plus the size of r; that of $r \bowtie\!\subset s$ is symmetric, while that of $r \supset\!\bowtie\!\subset s$ is the size of $r \bowtie s$ plus the sizes of r and s. All three estimates may be inaccurate, but provide upper bounds on the sizes.

14.2.5 Estimation of Number of Distinct Values

For selections, the number of distinct values of an attribute (or set of attributes) A in the result of a selection, $V(A, \sigma_\theta(r))$, can be estimated in these ways:

- If the selection condition θ forces A to take on a specified value (e.g., $A = 3$), $V(A, \sigma_\theta(r)) = 1$.
- If θ forces A to take on one of a specified set of values (e.g., $(A = 1 \vee A = 3 \vee A = 4)$), then $V(A, \sigma_\theta(r))$ is set to the number of specified values.
- If the selection condition θ is of the form $A\ op\ v$, where op is a comparison operator, $V(A, \sigma_\theta(r))$ is estimated to be $V(A, r) * s$, where s is the selectivity of the selection.
- In all other cases of selections, we assume that the distribution of A values is independent of the distribution of the values on which selection conditions are specified, and use an approximate estimate of $\min(V(A, r), n_{\sigma_\theta(r)})$. A more accurate estimate can be derived for this case using probability theory, but the above approximation works fairly well.

For joins, the number of distinct values of an attribute (or set of attributes) A in the result of a join, $V(A, r \bowtie s)$, can be estimated in these ways:

- If all attributes in A are from r, $V(A, r \bowtie s)$ is estimated as $\min(V(A, r), n_{r \bowtie s})$, and similarly if all attributes in A are from s, $V(A, r \bowtie s)$ is estimated to be $\min(V(A, s), n_{r \bowtie s})$.
- If A contains attributes $A1$ from r and $A2$ from s, then $V(A, r \bowtie s)$ is estimated as

  $$\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \bowtie s})$$

  Note that some attributes may be in $A1$ as well as in $A2$, and $A1 - A2$ and $A2 - A1$ denote, respectively, attributes in A that are only from r and attributes

in A that are only from s. Again, more accurate estimates can be derived by using probability theory, but the above approximations work fairly well.

The estimates of distinct values are straightforward for projections: They are the same in $\Pi_A(r)$ as in r. The same holds for grouping attributes of aggregation. For results of **sum**, **count**, and **average**, we can assume, for simplicity, that all aggregate values are distinct. For **min**(A) and **max**(A), the number of distinct values can be estimated as $\min(V(A, r), V(G, r))$, where G denotes the grouping attributes. We omit details of estimating distinct values for other operations.

14.3 Transformation of Relational Expressions

So far, we have studied algorithms to evaluate extended relational-algebra operations, and have estimated their costs. As mentioned at the start of this chapter, a query can be expressed in several different ways, with different costs of evaluation. In this section, rather than take the relational expression as given, we consider alternative, equivalent expressions.

Two relational-algebra expressions are said to be **equivalent** if, on every legal database instance, the two expressions generate the same set of tuples. (Recall that a legal database instance is one that satisfies all the integrity constraints specified in the database schema.) Note that the order of the tuples is irrelevant; the two expressions may generate the tuples in different orders, but would be considered equivalent as long as the set of tuples is the same.

In SQL, the inputs and outputs are multisets of tuples, and a multiset version of the relational algebra is used for evaluating SQL queries. Two expressions in the *multiset* version of the relational algebra are said to be equivalent if on every legal database the two expressions generate the same multiset of tuples. The discussion in this chapter is based on the relational algebra. We leave extensions to the multiset version of the relational algebra to you as exercises.

14.3.1 Equivalence Rules

An **equivalence rule** says that expressions of two forms are equivalent. We can replace an expression of the first form by an expression of the second form, or vice versa—that is we can replace an expression of the second form by an expression of the first form—since the two expressions would generate the same result on any valid database. The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.

We now list a number of general equivalence rules on relational-algebra expressions. Some of the equivalences listed appear in Figure 14.2. We use $\theta, \theta_1, \theta_2$, and so on to denote predicates, L_1, L_2, L_3, and so on to denote lists of attributes, and E, E_1, E_2, and so on to denote relational-algebra expressions. A relation name r is simply a special case of a relational-algebra expression, and can be used wherever E appears.

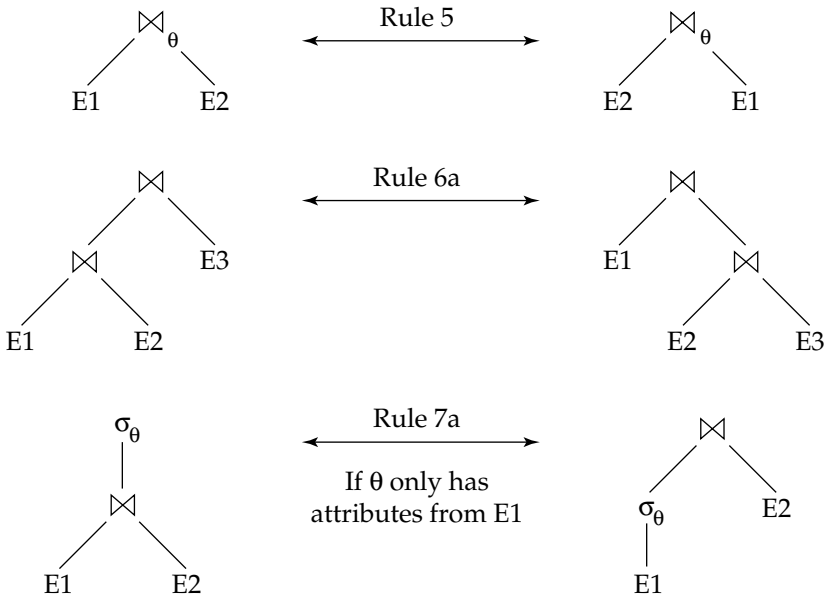


Figure 14.2 Pictorial representation of equivalences.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are **commutative**.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the final operations in a sequence of projection operations are needed, the others can be omitted. This transformation can also be referred to as a cascade of Π.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
      This expression is just the definition of the theta join.
   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Theta-join operations are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

   Actually, the order of attributes differs between the left-hand side and right-hand side, so the equivalence does not hold if the order of attributes is taken into account. A projection operation can be added to one of the sides of the equivalence to appropriately reorder attributes, but for simplicity we omit the projection and ignore the attribute order in most of our examples.

Recall that the natural-join operator is simply a special case of the theta-join operator; hence, natural joins are also commutative.

6. **a.** Natural-join operations are **associative**.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

b. Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3. Any of these conditions may be empty; hence, it follows that the Cartesian product ($\times$) operation is also associative. The commutativity and associativity of join operations are important for join reordering in query optimization.

7. The selection operation distributes over the theta-join operation under the following two conditions:

a. It distributes when all the attributes in selection condition θ_0 involve only the attributes of one of the expressions (say, E_1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

b. It distributes when selection condition θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over the theta-join operation under the following conditions.

a. Let L_1 and L_2 be attributes of E_1 and E_2, respectively. Suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

b. Consider a join $E_1 \bowtie_\theta E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively. Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

Set difference is not commutative.

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over the union, intersection, and set–difference operations.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

Similarly, the preceding equivalence, with $-$ replaced with either $\cup$ or $\cap$, also holds. Further,

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

The preceding equivalence, with $-$ replaced by $\cap$, also holds, but does not hold if $-$ is replaced by $\cup$.

12. The projection operation distributes over the union operation.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

This is only a partial list of equivalences. More equivalences involving extended relational operators, such as the outer join and aggregation, are discussed in the exercises.

14.3.2 Examples of Transformations

We now illustrate the use of the equivalence rules. We use our bank example with the relation schemas:

Branch-schema = (*branch-name, branch-city, assets*)
Account-schema = (*account-number, branch-name, balance*)
Depositor-schema = (*customer-name, account-number*)

The relations *branch*, *account*, and *depositor* are instances of these schemas.

In our example in Section 14.1, the expression

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch \bowtie (account \bowtie depositor)))$$

was transformed into the following expression,

$$\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch)) \bowtie (account \bowtie depositor))$$

which is equivalent to our original algebra expression, but generates smaller intermediate relations. We can carry out this transformation by using rule 7.a. Remember that the rule merely says that the two expressions are equivalent; it does not say that one is better than the other.

Multiple equivalence rules can be used, one after the other, on a query or on parts of the query. As an illustration, suppose that we modify our original query to restrict attention to customers who have a balance over $1000. The new relational-algebra query is

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie (account \bowtie depositor)))$$

We cannot apply the selection predicate directly to the *branch* relation, since the predicate involves attributes of both the *branch* and *account* relation. However, we can first

apply rule 6.a (associativity of natural join) to transform the join $branch \bowtie (account \bowtie depositor)$ into $(branch \bowtie account) \bowtie depositor$:

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}city = \text{“Brooklyn”}\ \wedge\ balance\ >1000} ((branch \bowtie\ account) \bowtie\ depositor))$$

Then, using rule 7.a, we can rewrite our query as

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{“Brooklyn”}\wedge\ balance>1000} (branch \bowtie\ account)) \bowtie\ depositor)$$

Let us examine the selection subexpression within this expression. Using rule 1, we can break the selection into two selections, to get the following subexpression:

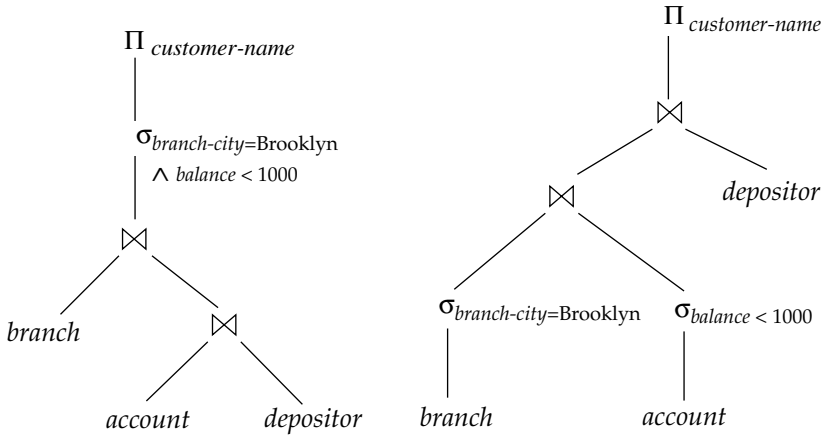
$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (\sigma_{balance\ >\ 1000}\ (branch \bowtie\ account))$$

Both of the preceding expressions select tuples with *branch-city* $=$ “Brooklyn” and *balance* > 1000. However, the latter form of the expression provides a new opportunity to apply the “perform selections early” rule, resulting in the subexpression

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch) \bowtie\ \sigma_{balance>1000}\ (account)$$

Figure 14.3 depicts the initial expression and the final expression after all these transformations. We could equally well have used rule 7.b to get the final expression directly, without using rule 1 to break the selection into two selections. In fact, rule 7.b can itself be derived from rules 1 and 7.a

A set of equivalence rules is said to be **minimal** if no rule can be derived from any combination of the others. The preceding example illustrates that the set of equivalence rules in Section 14.3.1 is not minimal. An expression equivalent to the original expression may be generated in different ways; the number of different ways of generating an expression increases when we use a nonminimal set of equivalence rules. Query optimizers therefore use minimal sets of equivalence rules.



(a) Initial expression tree (b) Tree after multiple transformations

Figure 14.3 Multiple transformations.

Now consider the following form of our example query:

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch) \bowtie\ account) \bowtie\ depositor)$$

When we compute the subexpression

$$(\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch) \bowtie\ account)$$

we obtain a relation whose schema is

(*branch-name, branch-city, assets, account-number, balance*)

We can eliminate several attributes from the schema, by pushing projections based on equivalence rules 8.a and 8.b. The only attributes that we must retain are those that either appear in the result of the query or are needed to process subsequent operations. By eliminating unneeded attributes, we reduce the number of columns of the intermediate result. Thus, we reduce the size of the intermediate result. In our example, the only attribute we need from the join of *branch* and *account* is *account-number*. Therefore, we can modify the expression to

$$\Pi_{customer\text{-}name}\ (\ (\ \Pi_{account\text{-}number}\ ((\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch)) \bowtie\ account)) \bowtie\ depositor)$$

The projection $\Pi_{account\text{-}number}$ reduces the size of the intermediate join results.

14.3.3 Join Ordering

A good ordering of join operations is important for reducing the size of temporary results; hence, most query optimizers pay a lot of attention to the join order. As mentioned in Chapter 3 and in equivalence rule 6.a, the natural-join operation is associative. Thus, for all relations $r_1,\ r_2$, and r_3,

$$(r_1 \bowtie\ r_2) \bowtie\ r_3\ =\ r_1 \bowtie\ (r_2 \bowtie\ r_3)$$

Although these expressions are equivalent, the costs of computing them may differ. Consider again the expression

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch)) \bowtie\ account \bowtie\ depositor)$$

We could choose to compute *account* $\bowtie$ *depositor* first, and then to join the result with

$$\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch)$$

However, *account* $\bowtie$ *depositor* is likely to be a large relation, since it contains one tuple for every account. In contrast,

$$\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch) \bowtie\ account$$

is probably a small relation. To see that it is, we note that, since the bank has a large number of widely distributed branches, it is likely that only a small fraction of the bank's customers have accounts in branches located in Brooklyn. Thus, the preceding expression results in one tuple for each account held by a resident of Brooklyn. Therefore, the temporary relation that we must store is smaller than it would have been had we computed *account* $\bowtie$ *depositor* first.

There are other options to consider for evaluating our query. We do not care about the order in which attributes appear in a join, since it is easy to change the order before displaying the result. Thus, for all relations r_1 and r_2,

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

That is, natural join is commutative (equivalence rule 5).

Using the associativity and commutativity of the natural join (rules 5 and 6), we can consider rewriting our relational-algebra expression as

$$\Pi_{customer\text{-}name}\ (((\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch)) \bowtie depositor) \bowtie account)$$

That is, we could compute

$$(\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch)) \bowtie depositor$$

first, and, after that, join the result with *account*. Note, however, that there are no attributes in common between *Branch-schema* and *Depositor-schema*, so the join is just a Cartesian product. If there are b branches in Brooklyn and d tuples in the *depositor* relation, this Cartesian product generates $b * d$ tuples, one for every possible pair of depositor tuple and branches (without regard for whether the account in *depositor* is maintained at the branch). Thus, it appears that this Cartesian product will produce a large temporary relation. As a result, we would reject this strategy. However, if the user had entered the preceding expression, we could use the associativity and commutativity of the natural join to transform this expression to the more efficient expression that we used earlier.

14.3.4 Enumeration of Equivalent Expressions

Query optimizers use equivalence rules to systematically generate expressions equivalent to the given query expression. Conceptually, the process proceeds as follows. Given an expression, if any subexpression matches one side of an equivalence rule, the optimizer generates a new expression where the subexpression is transformed to match the other side of the rule. This process continues until no more new expressions can be generated.

The preceding process is costly both in space and in time. Here is how the space requirement can be reduced: If we generate an expression E_1 from an expression E_2 by using an equivalence rule, then E_1 and E_2 are similar in structure, and have subexpressions that are identical. Expression-representation techniques that allow both expressions to point to shared subexpressions can reduce the space requirement significantly, and many query optimizers use them.

Moreover, it is not always necessary to generate every expression that can be generated with the equivalence rules. If an optimizer takes cost estimates of evaluation into account, it may be able to avoid examining some of the expressions, as we shall see in Section 14.4. We can reduce the time required for optimization by using techniques such as these.

14.4 Choice of Evaluation Plans

Generation of expressions is only part of the query-optimization process, since each operation in the expression can be implemented with different algorithms. An evaluation plan is therefore needed to define exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated. Figure 14.4 illustrates one possible evaluation plan for the expression from Figure 14.3. As we have seen, several different algorithms can be used for each relational operation, giving rise to alternative evaluation plans. Further, decisions about pipelining have to be made. In the figure, the edges from the selection operations to the merge join operation are marked as pipelined; pipelining is feasible if the selection operations generate their output sorted on the join attributes. They would do so if the indices on *branch* and *account* store records with equal values for the index attributes sorted by *branch-name*.

14.4.1 Interaction of Evaluation Techniques

One way to choose an evaluation plan for a query expression is simply to choose for each operation the cheapest algorithm for evaluating it. We can choose any ordering of the operations that ensures that operations lower in the tree are executed before operations higher in the tree.

However, choosing the cheapest algorithm for each operation independently is not necessarily a good idea. Although a merge join at a given level may be costlier than a hash join, it may provide a sorted output that makes evaluating a later operation (such as duplicate elimination, intersection, or another merge join) cheaper. Similarly, a nested-loop join with indexing may provide opportunities for pipelining the results to the next operation, and thus may be useful even if it is not the cheapest way of

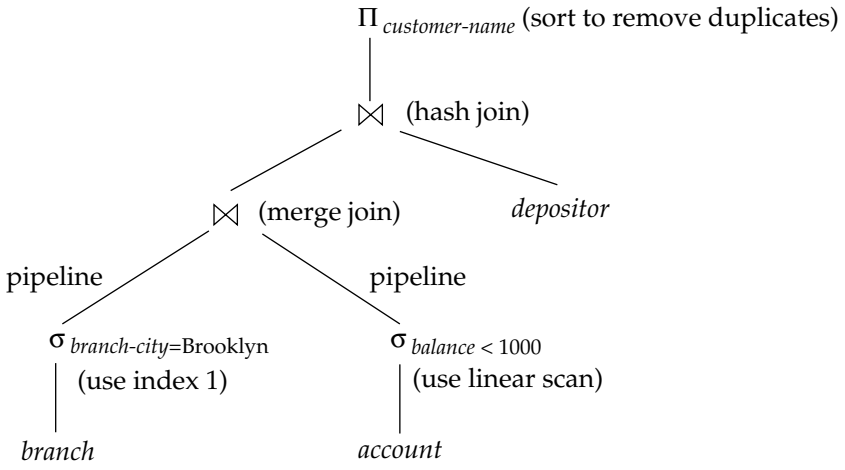


Figure 14.4 An evaluation plan.

performing the join. To choose the best overall algorithm, we must consider even nonoptimal algorithms for individual operations.

Thus, in addition to considering alternative expressions for a query, we must also consider alternative algorithms for each operation in an expression. We can use rules much like the equivalence rules to define what algorithms can be used for each operation, and whether its result can be pipelined or must be materialized. We can use these rules to generate all the query-evaluation plans for a given expression.

Given an evaluation plan, we can estimate its cost using statistics estimated by the techniques in Section 14.2 coupled with cost estimates for various algorithms and evaluation methods described in Chapter 13. That still leaves the problem of choosing the best evaluation plan for a query. There are two broad approaches: The first searches all the plans, and chooses the best plan in a cost-based fashion. The second uses heuristics to choose a plan. We discuss these approaches next. Practical query optimizers incorporate elements of both approaches.

14.4.2 Cost-Based Optimization

A **cost-based optimizer** generates a range of query-evaluation plans from the given query by using the equivalence rules, and chooses the one with the least cost. For a complex query, the number of different query plans that are equivalent to a given plan can be large. As an illustration, consider the expression

$$r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

where the joins are expressed without any ordering. With $n = 3$, there are 12 different join orderings:

$$\begin{array}{llll} r_1 \bowtie (r_2 \bowtie r_3) & r_1 \bowtie (r_3 \bowtie r_2) & (r_2 \bowtie r_3) \bowtie r_1 & (r_3 \bowtie r_2) \bowtie r_1 \\ r_2 \bowtie (r_1 \bowtie r_3) & r_2 \bowtie (r_3 \bowtie r_1) & (r_1 \bowtie r_3) \bowtie r_2 & (r_3 \bowtie r_1) \bowtie r_2 \\ r_3 \bowtie (r_1 \bowtie r_2) & r_3 \bowtie (r_2 \bowtie r_1) & (r_1 \bowtie r_2) \bowtie r_3 & (r_2 \bowtie r_1) \bowtie r_3 \end{array}$$

In general, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders. (We leave the computation of this expression for you to do in Exercise 14.10.) For joins involving small numbers of relations, this number is acceptable; for example, with $n = 5$, the number is 1680. However, as n increases, this number rises quickly. With $n = 7$, the number is 665280; with $n = 10$, the number is greater than 17.6 billion!

Luckily, it is not necessary to generate all the expressions equivalent to a given expression. For example, suppose we want to find the best join order of the form

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

which represents all join orders where r_1, r_2, and r_3 are joined first (in some order), and the result is joined (in some order) with r_4 and r_5. There are 12 different join orders for computing $r_1 \bowtie r_2 \bowtie r_3$, and 12 orders for computing the join of this result with r_4 and r_5. Thus, there appear to be 144 join orders to examine. However, once we have found the best join order for the subset of relations $\{r_1, r_2, r_3\}$, we can use that order for further joins with r_4 and r_5, and can ignore all costlier join orders of $r_1 \bowtie r_2 \bowtie r_3$. Thus, instead of 144 choices to examine, we need to examine only 12 + 12 choices.

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    for each non-empty subset S1 of S such that S1 ≠ S
        P1 = findbestplan(S1)
        P2 = findbestplan(S − S1)
        A = best algorithm for joining results of P1 and P2
        cost = P1.cost + P2.cost + cost of A
        if cost < bestplan[S].cost
            bestplan[S].cost = cost
            bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                join results of P1 and P2 using A"
    return bestplan[S]
```

Figure 14.5 Dynamic programming algorithm for join order optimization.

Using this idea, we can develop a *dynamic-programming* algorithm for finding optimal join orders. Dynamic programming algorithms store results of computations and reuse them, a procedure that can reduce execution time greatly. A recursive procedure implementing the dynamic programming algorithm appears in Figure 14.5.

The procedure stores the evaluation plans it computes in an associative array $bestplan$, which is indexed by sets of relations. Each element of the associative array contains two components: the cost of the best plan of S, and the plan itself. The value of $bestplan[S].cost$ is assumed to be initialized to ∞ if $bestplan[S]$ has not yet been computed.

The procedure first checks if the best plan for computing the join of the given set of relations S has been computed already (and stored in the associative array $bestplan$); if so it returns the already computed plan. Otherwise, the procedure tries every way of dividing S into two disjoint subsets. For each division, the procedure recursively finds the best plans for each of the two subsets, and then computes the cost of the overall plan by using that division. The procedure picks the cheapest plan from among all the alternatives for dividing S into two sets. The cheapest plan and its cost are stored in the array $bestplan$, and returned by the procedure. The time complexity of the procedure can be shown to be $O(3^n)$ (see Exercise 14.11).

Actually, the order in which tuples are generated by the join of a set of relations is also important for finding the best overall join order, since it can affect the cost of further joins (for instance, if merge join is used). A particular sort order of the tuples is said to be an **interesting sort order** if it could be useful for a later operation. For instance, generating the result of $r_1 \bowtie r_2 \bowtie r_3$ sorted on the attributes common with r_4 or r_5 may be useful, but generating it sorted on the attributes common to only r_1 and r_2 is not useful. Using merge join for computing $r_1 \bowtie r_2 \bowtie r_3$ may be costlier than using some other join technique, but may provide an output sorted in an interesting sort order.

Hence, it is not sufficient to find the best join order for each subset of the set of n given relations. Instead, we have to find the best join order for each subset, for

each interesting sort order of the join result for that subset. The number of subsets of n relations is 2^n. The number of interesting sort orders is generally not large. Thus, about 2^n join expressions need to be stored. The dynamic-programming algorithm for finding the best join order can be easily extended to handle sort orders. The cost of the extended algorithm depends on the number of interesting orders for each subset of relations; since this number has been found to be small in practice, the cost remains at $O(3^n)$.

With $n = 10$, this number is around 59000, which is much better than the 17.6 billion different join orders. More important, the storage required is much less than before, since we need to store only one join order for each interesting sort order of each of 1024 subsets of $r_1, \ldots, r_{10}$. Although both numbers still increase rapidly with n, commonly occurring joins usually have less than 10 relations, and can be handled easily.

We can use several techniques to reduce further the cost of searching through a large number of plans. For instance, when examining the plans for an expression, we can terminate after we examine only a part of the expression, if we determine that the cheapest plan for that part is already costlier than the cheapest evaluation plan for a full expression examined earlier. Similarly, suppose that we determine that the cheapest way of evaluating a subexpression is costlier than the cheapest evaluation plan for a full expression examined earlier. Then, no full expression involving that subexpression needs to be examined. We can further reduce the number of evaluation plans that need to be considered fully by first making a heuristic guess of a good plan, and estimating that plan's cost. Then, only a few competing plans will require a full analysis of cost. These optimizations can reduce the overhead of query optimization significantly.

14.4.3 Heuristic Optimization

A drawback of cost-based optimization is the cost of optimization itself. Although the cost of query processing can be reduced by clever optimizations, cost-based optimization is still expensive. Hence, many systems use **heuristics** to reduce the number of choices that must be made in a cost-based fashion. Some systems even choose to use only heuristics, and do not use cost-based optimization at all.

An example of a heuristic rule is the following rule for transforming relational-algebra queries:

- Perform selection operations as early as possible.

A heuristic optimizer would use this rule without finding out whether the cost is reduced by this transformation. In the first transformation example in Section 14.3, the selection operation was pushed into a join.

We say that the preceding rule is a heuristic because it usually, but not always, helps to reduce the cost. For an example of where it can result in an increase in cost, consider an expression $\sigma_\theta(r \bowtie s)$, where the condition θ refers to only attributes in s. The selection can certainly be performed before the join. However, if r is extremely small compared to s, and if there is an index on the join attributes of s, but no index on the attributes used by θ, then it is probably a bad idea to perform the selection

early. Performing the selection early—that is, directly on s—would require doing a scan of all tuples in s. It is probably cheaper, in this case, to compute the join by using the index, and then to reject tuples that fail the selection.

The projection operation, like the selection operation, reduces the size of relations. Thus, whenever we need to generate a temporary relation, it is advantageous to apply immediately any projections that are possible. This advantage suggests a companion to the "perform selections early" heuristic:

- Perform projections early.

It is usually better to perform selections earlier than projections, since selections have the potential to reduce the sizes of relations greatly, and selections enable the use of indices to access tuples. An example similar to the one used for the selection heuristic should convince you that this heuristic does not always reduce the cost.

Drawing on the equivalences discussed in Section 14.3.1, a heuristic optimization algorithm will reorder the components of an initial query tree to achieve improved query execution. We now present an overview of the steps in a typical heuristic optimization algorithm. You can understand the heuristics by visualizing a query expression as a tree, as illustrated in Figure 14.3

1. Deconstruct conjunctive selections into a sequence of single selection operations. This step, based on equivalence rule 1, facilitates moving selection operations down the query tree.

2. Move selection operations down the query tree for the earliest possible execution. This step uses the commutativity and distributivity properties of the selection operation noted in equivalence rules 2, 7.a, 7.b, and 11.

   For instance, this step transforms $\sigma_\theta(r \bowtie s)$ into either $\sigma_\theta(r) \bowtie s$ or $r \bowtie \sigma_\theta(s)$ whenever possible. Performing value-based selections as early as possible reduces the cost of sorting and merging intermediate results. The degree of reordering permitted for a particular selection is determined by the attributes involved in that selection condition.

3. Determine which selection operations and join operations will produce the smallest relations—that is, will produce the relations with the least number of tuples. Using associativity of the $\bowtie$ operation, rearrange the tree so that the leaf-node relations with these restrictive selections are executed first.

   This step considers the selectivity of a selection or join condition. Recall that the most restrictive selection—that is, the condition with the smallest selectivity—retrieves the fewest records. This step relies on the associativity of binary operations given in equivalence rule 6.

4. Replace with join operations those Cartesian product operations that are followed by a selection condition (rule 4.a). The Cartesian product operation is often expensive to implement since $r_1 \times r_2$ includes a record for each combination of records from r_1 and r_2. The selection may significantly reduce the number of records, making the join much less expensive than the Cartesian product.

5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed. This step draws on the properties of the projection operation given in equivalence rules 3, 8.a, 8.b, and 12.

6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining.

In summary, the heuristics listed here reorder an initial query-tree representation in such a way that the operations that reduce the size of intermediate results are applied first; early selection reduces the number of tuples, and early projection reduces the number of attributes. The heuristic transformations also restructure the tree so that the system performs the most restrictive selection and join operations before other similar operations.

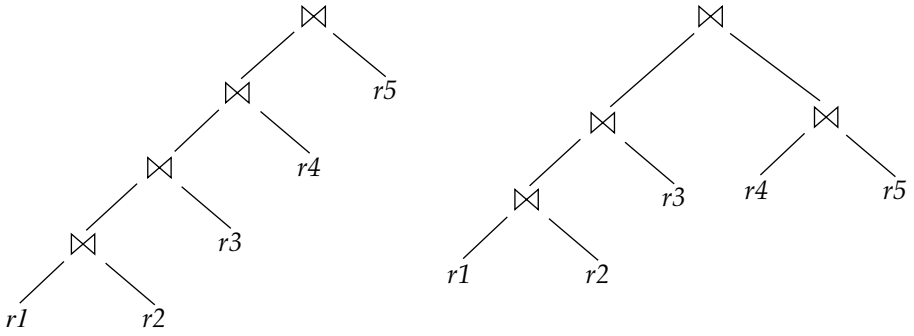
Heuristic optimization further maps the heuristically transformed query expression into alternative sequences of operations to produce a set of candidate evaluation plans. An evaluation plan includes not only the relational operations to be performed, but also the indices to be used, the order in which tuples are to be accessed, and the order in which the operations are to be performed. The **access-plan–selection** phase of a heuristic optimizer chooses the most efficient strategy for each operation.

14.4.4 Structure of Query Optimizers$**$

So far, we have described the two basic approaches to choosing an evaluation plan; as noted, most practical query optimizers combine elements of both approaches. For example, certain query optimizers, such as the System R optimizer, do not consider all join orders, but rather restrict the search to particular kinds of join orders. The System R optimizer considers only those join orders where the right operand of each join is one of the initial relations $r_1, \ldots, r_n$. Such join orders are called **left-deep join order**s. Left-deep join orders are particularly convenient for pipelined evaluation, since the right operand is a stored relation, and thus only one input to each join is pipelined.

Figure 14.6 illustrates the difference between left-deep join trees and non-left-deep join trees. The time it takes to consider all left-deep join orders is $O(n!)$, which is much less than the time to consider all join orders. With the use of dynamic programming optimizations, the System R optimizer can find the best join order in time $O(n2^n)$. Contrast this cost with the $O(3^n)$ time required to find the best overall join order. The System R optimizer uses heuristics to push selections and projections down the query tree.

The cost estimate that we presented for scanning by secondary indices assumed that every tuple access results in an I/O operation. The estimate is likely to be accurate with small buffers; with large buffers, however, the page containing the tuple may already be in the buffer. Some optimizers incorporate a better cost-estimation technique for such scans: They take into account the probability that the page containing the tuple is in the buffer.



(a) Left-deep join tree (b) Non-left-deep join tree

Figure 14.6 Left-deep join trees.

Query optimization approaches that integrate heuristic selection and the generation of alternative access plans have been adopted in several systems. The approach used in System R and in its successor, the Starburst project, is a hierarchical procedure based on the nested-block concept of SQL. The cost-based optimization techniques described here are used for each block of the query separately.

The heuristic approach in some versions of Oracle works roughly this way: For an n-way join, it considers n evaluation plans. Each plan uses a left-deep join order, starting with a different one of the n relations. The heuristic constructs the join order for each of the n evaluation plans by repeatedly selecting the "best" relation to join next, on the basis of a ranking of the available access paths. Either nested-loop or sort–merge join is chosen for each of the joins, depending on the available access paths. Finally, the heuristic chooses one of the n evaluation plans in a heuristic manner, based on minimizing the number of nested-loop joins that do not have an index available on the inner relation, and on the number of sort–merge joins.

The intricacies of SQL introduce a good deal of complexity into query optimizers. In particular, it is hard to translate nested subqueries in SQL into relational algebra. We briefly outline how to handle nested subqueries in Section 14.4.5. For compound SQL queries (using the $\cup$, $\cap$, or $-$ operation), the optimizer processes each component separately, and combines the evaluation plans to form the overall evaluation plan.

Even with the use of heuristics, cost-based query optimization imposes a substantial overhead on query processing. However, the added cost of cost-based query optimization is usually more than offset by the saving at query-execution time, which is dominated by slow disk accesses. The difference in execution time between a good plan and a bad one may be huge, making query optimization essential. The achieved saving is magnified in those applications that run on a regular basis, where the query can be optimized once, and the selected query plan can be used on each run. Therefore, most commercial systems include relatively sophisticated optimizers. The bibliographical notes give references to descriptions of the query optimizers of actual database systems.

14.4.5 Optimizing Nested Subqueries**

SQL conceptually treats nested subqueries in the **where** clause as functions that take parameters and return either a single value or a set of values (possibly an empty set). The parameters are the variables from outer level query that are used in the nested subquery (these variables are called **correlation variables**). For instance, suppose we have the following query.

```
select customer-name
from borrower
where exists (select *
              from depositor
              where depositor.customer-name = borrower.customer-name)
```

Conceptually, the subquery can be viewed as a function that takes a parameter (here, *borrower.customer-name*) and returns the set of all depositors with the same name.

SQL evaluates the overall query (conceptually) by computing the Cartesian product of the relations in the outer **from** clause and then testing the predicates in the **where** clause for each tuple in the product. In the preceding example, the predicate tests if the result of the subquery evaluation is empty.

This technique for evaluating a query with a nested subquery is called **correlated evaluation**. Correlated evaluation is not very efficient, since the subquery is separately evaluated for each tuple in the outer level query. A large number of random disk I/O operations may result.

SQL optimizers therefore attempt to transform nested subqueries into joins, where possible. Efficient join algorithms help avoid expensive random I/O. Where the transformation is not possible, the optimizer keeps the subqueries as separate expressions, optimizes them separately, and then evaluates them by correlated evaluation.

As an example of transforming a nested subquery into a join, the query in the preceding example can be rewritten as

```
select customer-name
from borrower, depositor
where depositor.customer-name = borrower.customer-name
```

(To properly reflect SQL semantics, the number of duplicate derivations should not change because of the rewriting; the rewritten query can be modified to ensure this property, as we will see shortly.)

In the example, the nested subquery was very simple. In general, it may not be possible to directly move the nested subquery relations into the **from** clause of the outer query. Instead, we create a temporary relation that contains the results of the nested query *without* the selections using correlation variables from the outer query, and join the temporary table with the outer level query. For instance, a query of the form

```
select ...
from L1
where P1 and exists (select *
                     from L2
                     where P2)
```

where P_2 is a conjunction of simpler predicates, can be rewritten as

```
create table t1 as
    select distinct V
        from L2
        where P2^1
select ...
from L1, t1
where P1 and P2^2
```

where P_2^1 contains predicates in P_2 without selections involving correlation variables, and P_2^2 reintroduces the selections involving correlation variables (with relations referenced in the predicate appropriately renamed). Here, V contains all attributes that are used in selections with correlation variables in the nested subquery.

In our example, the original query would have been transformed to

```
create table t1 as
    select distinct customer-name
    from depositor
select customer-name
from borrower, t1
where t1.customer-name = borrower.customer-name
```

The query we rewrote to illustrate creation of a temporary relation can be obtained by simplifying the above transformed query, assuming the number of duplicates of each tuple does not matter.

The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.

Decorrelation is more complicated when the nested subquery uses aggregation, or when the result of the nested subquery is used to test for equality, or when the condition linking the nested subquery to the outer query is **not exists**, and so on. We do not attempt to give algorithms for the general case, and instead refer you to relevant items in the bibliographical notes.

Optimization of complex nested subqueries is a difficult task, as you can infer from the above discussion, and many optimizers do only a limited amount of decorrelation. It is best to avoid using complex nested subqueries, where possible, since we cannot be sure that the query optimizer will succeed in converting them to a form that can be evaluated efficiently.

14.5 Materialized Views∗∗

When a view is defined, normally the database stores only the query defining the view. In contrast, a **materialized view** is a view whose contents are computed and stored. Materialized views constitute redundant data, in that their contents can be inferred from the view definition and the rest of the database contents. However, it is much cheaper in many cases to read the contents of a materialized view than to compute the contents of the view by executing the query defining the view.

Materialized views are important for improving performance in some applications. Consider this view, which gives the total loan amount at each branch:

create view *branch-total-loan*(*branch-name*, *total-loan*) **as**
select *branch-name*, **sum**(*amount*)
from *loan*
groupby *branch-name*

Suppose the total loan amount at the branch is required frequently (before making a new loan, for example). Computing the view requires reading every *loan* tuple pertaining to the branch, and summing up the loan amounts, which can be time-consuming.

In contrast, if the view definition of the total loan amount were materialized, the total loan amount could be found by looking up a single tuple in the materialized view.

14.5.1 View Maintenance

A problem with materialized views is that they must be kept up-to-date when the data used in the view definition changes. For instance, if the *amount* value of a loan is updated, the materialized view would become inconsistent with the underlying data, and must be updated. The task of keeping a materialized view up-to-date with the underlying data is known as **view maintenance**.

Views can be maintained by manually written code: That is, every piece of code that updates the *amount* value of a loan can be modified to also update the total loan amount for the corresponding branch.

Another option for maintaining materialized views is to define triggers on insert, delete, and update of each relation in the view definition. The triggers must modify the contents of the materialized view, to take into account the change that caused the trigger to fire. A simplistic way of doing so is to completely recompute the materialized view on every update.

A better option is to modify only the affected parts of the materialized view, which is known as **incremental view maintenance**. We describe how to perform incremental view maintenance in Section 14.5.2.

Modern database systems provide more direct support for incremental view maintenance. Database system programmers no longer need to define triggers for view maintenance. Instead, once a view is declared to be materialized, the database system computes the contents of the view, and incrementally updates the contents when the underlying data changes.

14.5.2 Incremental View Maintenance

To understand how to incrementally maintain materialized views, we start off by considering individual operations, and then see how to handle a complete expression.

The changes to a relation that can cause a materialized view to become out-of-date are inserts, deletes, and updates. To simplify our description, we replace updates to a tuple by deletion of the tuple followed by insertion of the updated tuple. Thus, we need to consider only inserts and deletes. The changes (inserts and deletes) to a relation or expression are referred to as its **differential**.

14.5.2.1 Join Operation

Consider the materialized view $v = r \bowtie s$. Suppose we modify r by inserting a set of tuples denoted by i_r. If the old value of r is denoted by r^{old}, and the new value of r by r^{new}, $r^{new} = r^{old} \cup i_r$. Now, the old value of the view, v^{old} is given by $r^{old} \bowtie s$, and the new value v^{new} is given by $r^{new} \bowtie s$. We can rewrite $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$, which we can again rewrite as $(r^{old} \bowtie s) \cup (i_r \bowtie s)$. In other words,

$$v^{new} = v^{old} \cup (i_r \bowtie s)$$

Thus, to update the materialized view v, we simply need to add the tuples $i_r \bowtie s$ to the old contents of the materialized view. Inserts to s are handled in an exactly symmetric fashion.

Now suppose r is modified by deleting a set of tuples denoted by d_r. Using the same reasoning as above, we get

$$v^{new} = v^{old} - (d_r \bowtie s)$$

Deletes on s are handled in an exactly symmetric fashion.

14.5.2.2 Selection and Projection Operations

Consider a view $v = \sigma_\theta(r)$. If we modify r by inserting a set of tuples i_r, the new value of v can be computed as

$$v^{new} = v^{old} \cup \sigma_\theta(i_r)$$

Similarly, if r is modified by deleting a set of tuples d_r, the new value of v can be computed as

$$v^{new} = v^{old} - \sigma_\theta(d_r)$$

Projection is a more difficult operation with which to deal. Consider a materialized view $v = \Pi_A(r)$. Suppose the relation r is on the schema $R = (A, B)$, and r contains two tuples $(a, 2)$ and $(a, 3)$. Then, $\Pi_A(r)$ has a single tuple (a). If we delete the tuple $(a, 2)$ from r, we cannot delete the tuple (a) from $\Pi_A(r)$: If we did so, the result would be an empty relation, whereas in reality $\Pi_A(r)$ still has a single tuple (a). The reason is that the same tuple (a) is derived in two ways, and deleting one tuple from r removes only one of the ways of deriving (a); the other is still present.

This reason also gives us the intuition for solution: For each tuple in a projection such as $\Pi_A(r)$, we will keep a count of how many times it was derived.

When a set of tuples d_r is deleted from r, for each tuple t in d_r we do the following. Let $t.A$ denote the projection of t on the attribute A. We find $(t.A)$ in the materialized view, and decrease the count stored with it by 1. If the count becomes 0, $(t.A)$ is deleted from the materialized view.

Handling insertions is relatively straightforward. When a set of tuples i_r is inserted into r, for each tuple t in i_r we do the following. If $(t.A)$ is already present in the materialized view, we increase the count stored with it by 1. If not, we add $(t.A)$ to the materialized view, with the count set to 1.

14.5.2.3 Aggregation Operations

Aggregation operations proceed somewhat like projections. The aggregate operations in SQL are **count, sum, avg, min,** and **max**:

- **count**: Consider a materialized view $v = {}_A\mathcal{G}_{count(B)}(r)$, which computes the count of the attribute B, after grouping r by attribute A.

  When a set of tuples i_r is inserted into r, for each tuple t in i_r we do the following. We look for the group $t.A$ in the materialized view. If it is not present, we add $(t.A, 1)$ to the materialized view. If the group $t.A$ is present, we add 1 to the count of the group.

  When a set of tuples d_r is deleted from r, for each tuple t in d_r we do the following. We look for the group $t.A$ in the materialized view, and subtract 1 from the count for the group. If the count becomes 0, we delete the tuple for the group $t.A$ from the materialized view.

- **sum**: Consider a materialized view $v = {}_A\mathcal{G}_{sum(B)}(r)$.

  When a set of tuples i_r is inserted into r, for each tuple t in i_r we do the following. We look for the group $t.A$ in the materialized view. If it is not present, we add $(t.A, t.B)$ to the materialized view; in addition, we store a count of 1 associated with $(t.A, t.B)$, just as we did for projection. If the group $t.A$ is present, we add the value of $t.B$ to the aggregate value for the group, and add 1 to the count of the group.

  When a set of tuples d_r is deleted from r, for each tuple t in d_r we do the following. We look for the group $t.A$ in the materialized view, and subtract $t.B$ from the aggregate value for the group. We also subtract 1 from the count for the group, and if the count becomes 0, we delete the tuple for the group $t.A$ from the materialized view.

  Without keeping the extra count value, we would not be able to distinguish a case where the sum for a group is 0 from the case where the last tuple in a group is deleted.

- **avg**: Consider a materialized view $v = {}_A\mathcal{G}_{avg(B)}(r)$.

  Directly updating the average on an insert or delete is not possible, since it depends not only on the old average and the tuple being inserted/deleted, but also on the number of tuples in the group.

Instead, to handle the case of **avg**, we maintain the **sum** and **count** aggregate values as described earlier, and compute the average as the sum divided by the count.

- **min, max**: Consider a materialized view $v = {}_A\mathcal{G}_{min(B)}(r)$. (The case of **max** is exactly equivalent.)

  Handling insertions on r is straightforward. Maintaining the aggregate values **min** and **max** on deletions may be more expensive. For example, if the tuple corresponding to the minimum value for a group is deleted from r, we have to look at the other tuples of r that are in the same group to find the new minimum value.

14.5.2.4 Other Operations

The set operation *intersection* is maintained as follows. Given materialized view $v = r \cap s$, when a tuple is inserted in r we check if it is present in s, and if so we add it to v. If a tuple is deleted from r, we delete it from the intersection if it is present. The other set operations, *union* and *set difference*, are handled in a similar fashion; we leave details to you.

Outer joins are handled in much the same way as joins, but with some extra work. In the case of deletion from r we have to handle tuples in s that no longer match any tuple in r. In the case of insertion to r, we have to handle tuples in s that did not match any tuple in r. Again we leave details to you.

14.5.2.5 Handling Expressions

So far we have seen how to update incrementally the result of a single operation. To handle an entire expression, we can derive expressions for computing the incremental change to the result of each subexpression, starting from the smallest subexpressions.

For example, suppose we wish to incrementally update a materialized view $E_1 \bowtie E_2$ when a set of tuples i_r is inserted into relation r. Let us assume r is used in E_1 alone. Suppose the set of tuples to be inserted into E_1 is given by expression D_1. Then the expression $D_1 \bowtie E_2$ gives the set of tuples to be inserted into $E_1 \bowtie E_2$.

See the bibliographical notes for further details on incremental view maintenance with expressions.

14.5.3 Query Optimization and Materialized Views

Query optimization can be performed by treating materialized views just like regular relations. However, materialized views offer further opportunities for optimization:

- Rewriting queries to use materialized views:

  Suppose a materialized view $v = r \bowtie s$ is available, and a user submits a query $r \bowtie s \bowtie t$. Rewriting the query as $v \bowtie t$ may provide a more efficient query plan than optimizing the query as submitted. Thus, it is the job of the

query optimizer to recognize when a materialized view can be used to speed up a query.

- Replacing a use of a materialized view by the view definition:
  Suppose a materialized view $v = r \bowtie s$ is available, but without any index on it, and a user submits a query $\sigma_{A=10}(v)$. Suppose also that s has an index on the common attribute B, and r has an index on attribute A. The best plan for this query may be to replace v by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$; the selection and join can be performed efficiently by using the indices on $r.A$ and $s.B$, respectively. In contrast, evaluating the selection directly on v may require a full scan of v, which may be more expensive.

The bibliographical notes give pointers to research showing how to efficiently perform query optimization with materialized views.

Another related optimization problem is that of **materialized view selection**, namely, "What is the best set of views to materialize?" This decision must be made on the basis of the system **workload**, which is a sequence of queries and updates that reflects the typical load on the system. One simple criterion would be to select a set of materialized views that minimizes the overall execution time of the workload of queries and updates, including the time taken to maintain the materialized views. Database administrators usually modify this criterion to take into account the importance of different queries and updates: Fast response may be required for some queries and updates, but a slow response may be acceptable for others.

Indices are just like materialized views, in that they too are derived data, can speed up queries, and may slow down updates. Thus, the problem of **index selection** is closely related, to that of materialized view selection, although it is simpler.

We examine these issues in more detail in Sections 21.2.5 and 21.2.6.

Some database systems, such as Microsoft SQL Server 7.5, and the RedBrick Data Warehouse from Informix, provide tools to help the database administrator with index and materialized view selection. These tools examine the history of queries and updates, and suggest indices and views to be materialized.

14.6 Summary

- Given a query, there are generally a variety of methods for computing the answer. It is the responsibility of the system to transform the query as entered by the user into an equivalent query that can be computed more efficiently. The process of finding a good strategy for processing a query, is called *query optimization*.
- The evaluation of complex queries involves many accesses to disk. Since the transfer of data from disk is slow relative to the speed of main memory and the CPU of the computer system, it is worthwhile to allocate a considerable amount of processing to choose a method that minimizes disk accesses.
- The strategy that the database system chooses for evaluating an operation depends on the size of each relation and on the distribution of values within

columns. So that they can base the strategy choice on reliable information, database systems may store statistics for each relation r. These statistics include

- The number of tuples in the relation r
- The size of a record (tuple) of relation r in bytes
- The number of distinct values that appear in the relation r for a particular attribute

- These statistics allow us to estimate the sizes of the results of various operations, as well as the cost of executing the operations. Statistical information about relations is particularly useful when several indices are available to assist in the processing of a query. The presence of these structures has a significant influence on the choice of a query-processing strategy.
- Each relational-algebra expression represents a particular sequence of operations. The first step in selecting a query-processing strategy is to find a relational-algebra expression that is equivalent to the given expression and is estimated to cost less to execute.
- There are a number of equivalence rules that we can use to transform an expression into an equivalent one. We use these rules to generate systematically all expressions equivalent to the given query.
- Alternative evaluation plans for each expression can be generated by similar rules, and the cheapest plan across all expressions can be chosen. Several optimization techniques are available to reduce the number of alternative expressions and plans that need to be generated.
- We use heuristics to reduce the number of plans considered, and thereby to reduce the cost of optimization. Heuristic rules for transforming relational-algebra queries include "Perform selection operations as early as possible," "Perform projections early," and "Avoid Cartesian products."
- Materialized views can be used to speed up query processing. Incremental view maintenance is needed to efficiently update materialized views when the underlying relations are modified. The differential of an operation can be computed by means of algebraic expressions involving differentials of the inputs of the operation. Other issues related to materialized views include how to optimize queries by making use of available materialized views, and how to select views to be materialized.

Review Terms

- Query optimization
- Statistics estimation
- Catalog information
- Size estimation
  - Selection
  - Selectivity
  - Join

- Distinct value estimation
- Transformation of expressions
- Cost-based optimization
- Equivalence of expressions
- Equivalence rules
  - Join commutativity
  - Join associativity
- Minimal set of equivalence rules
- Enumeration of equivalent expressions
- Choice of evaluation plans
- Interaction of evaluation techniques
- Join order optimization
  - Dynamic-programming algorithm
  - Left-deep join order
- Heuristic optimization
- Access-plan selection
- Correlated evaluation
- Decorrelation
- Materialized views
- Materialized view maintenance
  - Recomputation
  - Incremental maintenance
  - Insertion,
  - Deletion
  - Updates
- Query optimization with materialized views
- Index selection
- Materialized view selection

Exercises

14.1 Clustering indices may allow faster access to data than a nonclustering index affords. When must we create a nonclustering index, despite the advantages of a clustering index? Explain your answer.

14.2 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$, with primary keys *A*, *C*, and *E*, respectively. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.

14.3 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$ of Exercise 14.2. Assume that there are no primary keys, except the entire schema. Let $V(C, r_1)$ be 900, $V(C, r_2)$ be 1100, $V(E, r_2)$ be 50, and $V(E, r_3)$ be 100. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.

14.4 Suppose that a B^+-tree index on *branch-city* is available on relation *branch*, and that no other index is available. What would be the best way to handle the following selections that involve negation?

a. $\sigma_{\neg(branch\text{-}city < \text{“Brooklyn”})}(branch)$
b. $\sigma_{\neg(branch\text{-}city = \text{“Brooklyn”})}(branch)$
c. $\sigma_{\neg(branch\text{-}city < \text{“Brooklyn”} \vee assets < 5000)}(branch)$

14.5 Suppose that a B^+-tree index on (*branch-name, branch-city*) is available on relation *branch*. What would be the best way to handle the following selection?

$$\sigma_{(branch\text{-}city < \text{“Brooklyn”}) \wedge (assets < 5000) \wedge (branch\text{-}name = \text{“Downtown”})}(branch)$$

14.6 Show that the following equivalences hold. Explain how you can apply then to improve the efficiency of certain queries:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.
b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, where θ uses only attributes from A.
c. $\sigma_\theta(E_1 \mathbin{⟕} E_2) = \sigma_\theta(E_1) \mathbin{⟕} E_2$ where θ uses only attributes from E_1.

14.7 Show how to derive the following equivalences by a sequence of transformations using the equivalence rules in Section 14.3.1.

a. $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$
b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, where θ_2 involves only attributes from E_2

14.8 For each of the following pairs of expressions, give instances of relations that show the expressions are not equivalent.

a. $\Pi_A(R - S)$ and $\Pi_A(R) - \Pi_A(S)$
b. $\sigma_{B<4}({}_A\mathcal{G}_{max(B)}(R))$ and ${}_A\mathcal{G}_{max(B)}(\sigma_{B<4}(R))$
c. In the preceding expressions, if both occurrences of max were replaced by min would the expressions be equivalent?
d. $(R \mathbin{⟕} S) \mathbin{⟕} T$ and $R \mathbin{⟕} (S \mathbin{⟕} T)$
In other words, the natural left outer join is not associative.
(Hint: Assume that the schemas of the three relations are $R(a, b1), S(a, b2)$, and $T(a, b3)$, respectively.)
e. $\sigma_\theta(E_1 \mathbin{⟕} E_2)$ and $E_1 \mathbin{⟕} \sigma_\theta(E_2)$, where θ uses only attributes from E_2

14.9 SQL allows relations with duplicates (Chapter 4).

a. Define versions of the basic relational-algebra operations σ, Π, $\times$, $\bowtie$, $-$, $\cup$, and $\cap$ that work on relations with duplicates, in a way consistent with SQL.
b. Check which of the equivalence rules 1 through 7.b hold for the multiset version of the relational-algebra defined in part a.

14.10 ∗∗ Show that, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders.

Hint: A **complete binary tree** is one where every internal node has exactly two children. Use the fact that the number of different complete binary trees with n leaf nodes is $\frac{1}{n}\binom{2(n-1)}{(n-1)}$.

If you wish, you can derive the formula for the number of complete binary trees with n nodes from the formula for the number of binary trees with n nodes. The number of binary trees with n nodes is $\frac{1}{n+1}\binom{2n}{n}$; this number is known as the Catalan number, and its derivation can be found in any standard textbook on data structures or algorithms.

14.11 ∗∗ Show that the lowest-cost join order can be computed in time $O(3^n)$. Assume that you can store and look up information about a set of relations (such as the optimal join order for the set, and the cost of that join order) in constant time. (If you find this exercise difficult, at least show the looser time bound of $O(2^{2n})$.)

14.12 Show that, if only left-deep join trees are considered, as in the System R optimizer, the time taken to find the most efficient join order is around $n2^n$. Assume that there is only one interesting sort order.

14.13 A set of equivalence rules is said to be *complete* if, whenever two expressions are equivalent, one can be derived from the other by a sequence of uses of the equivalence rules. Is the set of equivalence rules that we considered in Section 14.3.1 complete? Hint: Consider the equivalence $\sigma_{3=5}(r) = \{\ \}$.

14.14 Decorrelation:

a. Write a nested query on the relation *account* to find for each branch with name starting with "B", all accounts with the maximum balance at the branch.

b. Rewrite the preceding query, without using a nested subquery; in other words, decorrelate the query.

c. Give a procedure (similar that that described in Section 14.4.5) for decorrelating such queries.

14.15 Describe how to incrementally maintain the results of the following operations, on both insertions and deletions.

a. Union and set difference

b. Left outer join

14.16 Give an example of an expression defining a materialized view and two situations (sets of statistics for the input relations and the differentials) such that incremental view maintenance is better than recomputation in one situation, and recomputation is better in the other situation.

Bibliographical Notes

The seminal work of Selinger et al. [1979] describes access-path selection in the System R optimizer, which was one of the earliest relational-query optimizers. Graefe and McKenna [1993] describe Volcano, an equivalence-rule based query optimizer. Query processing in Starburst is described in Haas et al. [1989]. Query optimization in Oracle is briefly outlined in Oracle [1997].

Estimation of statistics of query results, such as result size, is addressed by Ioannidis and Poosala [1995], Poosala et al. [1996], and Ganguly et al. [1996], among others. Nonuniform distributions of values causes problems for estimation of query size and cost. Cost-estimation techniques that use histograms of value distributions have been proposed to tackle the problem. Ioannidis and Christodoulakis [1993], Ioannidis and Poosala [1995], and Poosala et al. [1996] present results in this area.

Exhaustive searching of all query plans is impractical for optimization of joins involving many relations, and techniques based on randomized searching, which do not examine all alternatives, have been proposed. Ioannidis and Wong [1987], Swami and Gupta [1988], and Ioannidis and Kang [1990] present results in this area.

Parametric query-optimization techniques have been proposed by Ioannidis et al. [1992] and Ganguly [1998], to handle query processing when the selectivity of query

parameters is not known at optimization time. A set of plans—one for each of several different query selectivities—is computed, and is stored by the optimizer, at compile time. One of these plans is chosen at run time, on the basis of the actual selectivities, avoiding the cost of full optimization at run time.

Klug [1982] was an early work on optimization of relational-algebra expressions with aggregate functions. More recent work in this area includes Yan and Larson [1995] and Chaudhuri and Shim [1994]. Optimization of queries containing outer joins is described in Rosenthal and Reiner [1984], Galindo-Legaria and Rosenthal [1992], and Galindo-Legaria [1994].

The SQL language poses several challenges for query optimization, including the presence of duplicates and nulls, and the semantics of nested subqueries. Extension of relational algebra to duplicates is described in Dayal et al. [1982]. Optimization of nested subqueries is discussed in Kim [1982], Ganski and Wong [1987], Dayal [1987], and more recently, in Seshadri et al. [1996].

When queries are generated through views, more relations often are joined than is necessary for computation of the query. A collection of techniques for join minimization has been grouped under the name *tableau optimization*. The notion of a tableau was introduced by Aho et al. [1979b] and Aho et al. [1979a], and was further extended by Sagiv and Yannakakis [1981]. Ullman [1988] andMaier [1983] provide a textbook coverage of tableaux.

Sellis [1988] and Roy et al. [2000] describe *multiquery optimization*, which is the problem of optimizing the execution of several queries as a group. If an entire group of queries is considered, it is possible to discover *common subexpressions* that can be evaluated once for the entire group. Finkelstein [1982] and Hall [1976] consider optimization of a group of queries and the use of common subexpressions. Dalvi et al. [2001] discuss optimization issues in pipelining with limited buffer space combined with sharing of common subexpressions.

Query optimization can make use of semantic information, such as functional dependencies and other integrity constraints. *Semantic query-optimization* in relational databases is covered by King [1981], Chakravarthy et al. [1990], and in the context of aggregation, by Sudarshan and Ramakrishnan [1991].

Query-processing and optimization techniques for Datalog, in particular techniques to handle queries on recursive views, are described in Bancilhon and Ramakrishnan [1986], Beeri and Ramakrishnan [1991], Ramakrishnan et al. [1992c], Srivastava et al. [1995] and Mumick et al. [1996]. Query processing and optimization techniques for object-oriented databases are discussed in Maier and Stein [1986], Beech [1988], Bertino and Kim [1989], and Blakeley et al. [1993].

Blakeley et al. [1986], Blakeley et al. [1989], and Griffin and Libkin [1995] describe techniques for maintenance of materialized views. Gupta and Mumick [1995] provides a survey of materialized view maintenance. Optimization of materialized view maintenance plans is described by Vista [1998] and Mistry et al. [2001]. Query optimization in the presence of materialized views is addressed by Larson and Yang [1985], Chaudhuri et al. [1995], Dar et al. [1996], and Roy et al. [2000]. Index selection and materialized view selection are addressed by Ross et al. [1996], Labio et al. [1997], Gupta [1997], Chaudhuri and Narasayya [1997], and Roy et al. [2000].