Chapter 10

Storage and File Structure

1. We have been looking mostly at the higher-level models of a database. At the **conceptual** or **logical** level the database was viewed as
   - A collection of tables (relational model).
   - A collection of classes of objects (object-oriented model).
2. The logical model is the correct level for database **users** to focus on. However, **performance** depends on the efficiency of the data structures used to represent data in the database, and on the efficiency of operations on these data structures.

10.1 Overview of Physical Storage Media

1. Several types of data storage exist in most computer systems. They vary in speed of access, cost per unit of data, and reliability.
   - **Cache:** most costly and fastest form of storage. Usually very small, and managed by the operating system.
   - **Main Memory (MM):** the storage area for data available to be operated on.
     - General-purpose machine instructions operate on main memory.
     - Contents of main memory are usually lost in a power failure or "crash".
     - Usually too small (even with megabytes) and too expensive to store the entire database.
   - **Flash memory:** EEPROM (*electrically erasable programmable read-only memory*).
     - Data in flash memory survive from power failure.
     - Reading data from flash memory takes about 10 nano-secs (roughly as fast as from main memory), and writing data into flash memory is more complicated: write-once takes about 4-10 microsecs.
     - To overwrite what has been written, one has to first erase the entire bank of the memory. It may support only a limited number of erase cycles (10^4 to 10^6).
     - It has found its popularity as a replacement for disks for storing small volumes of data (5-10 megabytes).
   - **Magnetic-disk storage:** primary medium for long-term storage.
     - Typically the entire database is stored on disk.
     - Data must be moved from disk to main memory in order for the data to be operated on.
     - After operations are performed, data must be copied back to disk if any changes were made.

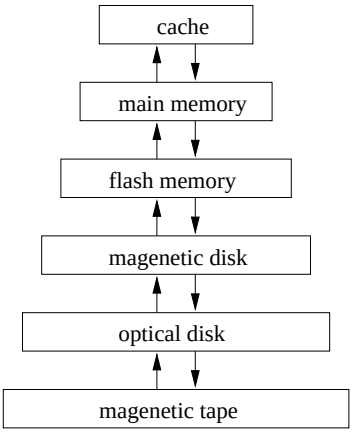

Figure 10.1: Storage-device hierarchy

  - Disk storage is called **direct access** storage as it is possible to read data on the disk in any order (unlike sequential access).
  - Disk storage usually survives power failures and system crashes.
- **Optical storage:** CD-ROM (compact-disk read-only memory), WORM (*write-once read-many*) disk (for archival storage of data), and *Juke box* (containing a few drives and numerous disks loaded on demand).
- **Tape Storage:** used primarily for backup and archival data.
  - Cheaper, but much slower access, since tape must be read sequentially from the beginning.
  - Used as protection from disk failures!

2. The storage device hierarchy is presented in Figure 10.1, where the higher levels are expensive (cost per bit), fast (access time), but the capacity is smaller.

3. Another classification: Primary, secondary, and tertiary storage.

   (a) Primary storage: the fastest storage media, such as cash and main memory.

   (b) Secondary (or on-line) storage: the next level of the hierarchy, e.g., magnetic disks.

   (c) Tertiary (or off-line) storage: magnetic tapes and optical disk juke boxes.

4. Volatility of storage. *Volatile storage* loses its contents when the power is removed. Without power backup, data in the volatile storage (the part of the hierarchy from main memory up) must be written to nonvolatile storage for safekeeping.

10.2 Magnetic Disks

10.2.1 Physical Characteristics of Disks

1. The storage capacity of a single disk ranges from 10MB to 10GB. A typical commercial database may require hundreds of disks.

2. Figure 10.2 shows a moving-head disk mechanism.

   - Each disk *platter* has a flat circular shape. Its two surfaces are covered with a magnetic material and information is recorded on the surfaces. The platter of *hard disks* are made from rigid metal or glass, while *floppy disks* are made from flexible material.

- The disk surface is logically divided into *tracks*, which are subdivided into *sectors*. A sector (varying from 32 bytes to 4096 bytes, usually 512 bytes) is the smallest unit of information that can be read from or written to disk. There are 4-32 sectors per track and 20-1500 tracks per disk surface.
- The **arm** can be positioned over any one of the tracks.
- The **platter** is spun at high speed.
- To read information, the arm is **positioned** over the correct track.
- When the data to be accessed passes under the head, the **read** or **write** operation is performed.

3. A disk typically contains multiple platters (see Figure 10.2). The read-write heads of all the tracks are mounted on a single assembly called a *disk arm*, and move together.
   - Multiple disk arms are moved as a unit by the **actuator**.
   - Each arm has two heads, to read disks above and below it.
   - The set of **tracks** over which the heads are located forms a **cylinder**.
   - This cylinder holds that data that is accessible within the disk latency time.
   - It is clearly sensible to store related data in the same or adjacent cylinders.
4. Disk platters range from 1.8" to 14" in diameter, and 5"1/4 and 3"1/2 disks dominate due to the lower cost and faster seek time than do larger disks, yet they provide high storage capacity.
5. A *disk controller* interfaces between the computer system and the actual hardware of the disk drive. It accepts commands to r/w a sector, and initiate actions. Disk controllers also attach *checksums* to each sector to check read error.
6. *Remapping of bad sectors*: If a 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.
7. SCSI (*Small Computer System Interconnect*) is commonly used to connect disks to PCs and workstations. Mainframe and server systems usually have a faster and more expensive bus to connect to the disks.
8. Head crash: why cause the entire disk failing (?).
9. A *fixed dead disk* has a separate head for each track — very many heads, very expensive. *Multiple disk arms*: allow more than one track to be accessed at a time. Both were used in high performance mainframe systems but are relatively rare today.

10.2.2 Performance Measures of Disks

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

1. *access time*: the time from when a read or write request is issued to when data transfer begins. To access 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 **seek time**, and it increases with the distance the arm must move. Typical seek time range from 2 to 30 milliseconds.

   Average seek time is the average of the seek time, measured over a sequence of (uniformly distributed) random requests, and it is about one third of the worst-case seek time.

   Once the seek has occurred, the time spent waiting for the sector to be accesses to appear under the head is called **rotational latency time**. Average rotational latency time is about half of the time for a full rotation of the disk. (Typical rotational speeds of disks ranges from 60 to 120 rotations per second).

   The access time is then the sum of the seek time and the latency and ranges from 10 to 40 milli-sec.
2. *data transfer rate*, the rate at which data can be retrieved from or stored to the disk. Current disk systems support transfer rate from 1 to 5 megabytes per second.
3. *reliability*, measured by the *mean time to failure*. The typical mean time to failure of disks today ranges from 30,000 to 800,000 hours (about 3.4 to 91 years).

10.2.3 Optimization of Disk-Block Access

1. Data is transferred between disk and main memory in units called **blocks**.
2. A **block** is a contiguous sequence of bytes from a single track of one platter.
3. Block sizes range from 512 bytes to several thousand.
4. The lower levels of file system manager covert block addresses into the hardware-level cylinder, surface, and sector number.
5. Access to data on disk is several orders of magnitude slower than is access to data in main memory. Optimization techniques besides buffering of blocks in main memory.

   - **Scheduling:** If several blocks from a cylinder need to be transferred, we may save time by requesting them in the order in which they pass under the heads. A commonly used disk-arm scheduling algorithm is the *elevator algorithm.*
   - **File organization**. Organize blocks on disk in a way that corresponds closely to the manner that we expect data to be accessed. For example, store related information on the same track, or physically close tracks, or adjacent cylinders in order to minimize seek time. IBM mainframe OS's provide programmers fine control on placement of files but increase programmer's burden.

     UNIX or PC OSs hide disk organizations from users. Over time, a sequential file may become fragmented. To reduce fragmentation, the system can make a back-up copy of the data on disk and restore the entire disk. The restore operation writes back the blocks of each file continuously (or nearly so). Some systems, such as MS-DOS, have utilities that scan the disk and then move blocks to decrease the fragmentation.
   - **Nonvolatile write buffers**. Use *nonvolatile RAM* (such as *battery-back-up RAM*) to speed up disk writes drastically (first write to nonvolatile RAM buffer and inform OS that writes completed).
   - **Log disk**. Another approach to reducing write latency is to use a *log disk*, a disk devoted to writing a sequential log. All access to the log disk is sequential, essentially eliminating seek time, and several consecutive blocks can be written at once, making writes to log disk several times faster than random writes.

10.3 RAID: Redundant Arrays of Inexpensive Disks (Not covered)

10.4 Tertiary Storage

10.4.1 Optical Disks

1. CD-ROM has become a popular medium for distributing software, multimedia data, and other electronic published information.
2. Capacity of CD-ROM: $\sim$ 500 MB. Disks are cheap to mass produce and also drives.
3. CD-ROM: much longer seek time (250m-sec), lower rotation speed (400 rpm), leading to high latency and lower data-transfer rate (about 150 KB/sec). Drives spins at 8/12$\times$ audio CD spin speed (standard) is available.
4. Recently, a new optical format, *digit video disk (DVD)* has become standard. These disks hold between 4.7 and 17 GB data.
5. WORM (write-once, read many) disks are popular for archival storage of data since they have a high capacity (about 500 MB), longer life time than HD, and can be removed from drive — good for audit trail (hard to tamper).

10.4.2 Magnetic Tapes

1. Long history, slow, and limited to sequential access, and thus are used for backup, storage for infrequent access, and off-line medium for system transfer.
2. Moving to the correct spot may take minutes, but once positioned, tape drives can write data at density and speed approaching to those of disk drives.
3. 8mm tape drive has the highest density, and we store 5 GB data on a 350-foot tape.
4. Popularly used for storage of large volumes of data, such as video, image, or remote sensing data.

10.5 Storage Access

1. Each file is partitioned into fixed-length storage units, called *blocks*, which are the units of both storage allocation and data transfer.
2. It is desirable to keep as many blocks as possible in main memory. Usually, we cannot keep all blocks in main memory, so we need to manage the allocation of available main memory space.
3. We need to use disk storage for the database, and to transfer blocks of data between main memory and disk. We also want to minimize the number of such transfers, as they are time-consuming.
4. The **buffer** is the part of main memory available for storage of **copies** of disk blocks.

10.5.1 Buffer manager

1. The subsystem responsible for the allocation of buffer space is called the **buffer manager**.
   - The buffer manager handles all requests for blocks of the database.
   - If the block is already in main memory, the address in main memory is given to the requester.
   - If not, the buffer manager must read the block in from disk (possibly displacing some other block if the buffer is full) and then pass the address in main memory to the requester.
2. The buffer manager must use some sophisticated techniques in order to provide good service:
   - **Replacement Strategy** — When there is no room left in the buffer, some block must be removed to make way for the new one. Typical operating system memory management schemes use a "least recently used" (**LRU**) method. (Simply remove the block least recently referenced.) This can be improved upon for database applications.
   - **Pinned Blocks** – For the database to be able to recover from crashes, we need to restrict times when a block maybe written back to disk. A block not allowed to be written is said to be **pinned**. Many operating systems do not provide support for pinned blocks, and such a feature is essential if a database is to be "crash resistant".
   - **Forced Output of Blocks** – Sometimes it is necessary to write a block back to disk even though its buffer space is not needed, (called the **forced output** of a block.) This is due to the fact that main memory contents (and thus the buffer) are lost in a crash, while disk data usually survives.

10.5.2 Buffer replacement policies

1. **Replacement Strategy:** Goal is minimization of accesses to disk. Generally it is hard to predict which blocks will be referenced. So operating systems use the history of past references as a guide to prediction.
   - **General Assumption:** Blocks referenced recently are likely to be used again.
   - Therefore, if we need space, throw out the least recently referenced block (LRU replacement scheme).

2. LRU is acceptable in **operating systems**, however, a database system is able to predict future references more accurately.

3. Consider processing of the relational algebra expression

$$borrow \bowtie customer$$

4. Further, assume the strategy to process this request is given by the following pseudo-code:

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

5. Assume that the two relations in this example are stored in different files.

   - Once a tuple of *borrower* has been processed, it is not needed again. Therefore, once processing of an entire block of tuples is finished, that block is not needed in main memory, even though it has been used **very** recently.
   - Buffer manager should free the space occupied by a borrow block as soon as it is processed. This strategy is called **toss-immediate**.
   - Consider blocks containing *customer* tuples.
   - Every block of *customer* tuples must be examined once for every tuple of the *borrow* relation. When processing of a *customer* block is completed, it will not be used again until all other *customer* blocks have been processed. This means the most recently used (MRU) block will be the last block to be re-referenced, and the least recently used will be referenced next.
   - This is the opposite of LRU assumptions. So for inner block, use MRU strategy — if a customer block must be removed from the buffer, choose MRU block.
   - For MRU strategy, the system must **pin** the *customer* block currently being processed until the last tuple has been processed. Then it is unpinned, becoming the most recently used block.

6. The buffer manager may also use statistical information regarding the probability that a request will reference a particular relation.

   - The data dictionary is the most frequently-used part of the database. It should, therefore, not be removed from main memory unless necessary.
   - File indices are also frequently used, and should generally be in main memory.
   - No single strategy is known that handles all possible scenarios well.
   - Many database systems use LRU, despite of its faults.
   - Concurrency and recovery may need other buffer management strategies, such as delayed buffer-out or forced output.

10.6 File Organization

1. A **file** is organized logically as a sequence of records.
2. Records are mapped onto disk blocks.
3. Files are provided as a basic construct in operating systems, so we assume the existence of an underlying **file system**.
4. Blocks are of a fixed size determined by the operating system.
5. Record sizes vary.
6. In relational database, tuples of distinct relations may be of different sizes.
7. One approach to mapping database to files is to store records of one length in a given file.
8. An alternative is to structure files to accommodate variable-length records. (Fixed-length is easier to implement.)

10.6.1 Fixed-Length Records

1. Consider a file of deposit records of the form:

```
type deposit = record
        bname : char(22);
        account# : char(10);
        balance : real;
    end
```

   - If we assume that each character occupies one byte, an integer occupies 4 bytes, and a real 8 bytes, our deposit record is 40 bytes long.
   - The simplest approach is to use the first 40 bytes for the first record, the next 40 bytes for the second, and so on.
   - However, there are two problems with this approach.
   - It is difficult to delete a record from this structure.
   - Space occupied must somehow be deleted, or we need to mark deleted records so that they can be ignored.
   - Unless block size is a multiple of 40, some records will cross block boundaries.
   - It would then require two block accesses to read or write such a record.

2. When a record is deleted, we could move all successive records up one (Figure 10.7), which may require moving a lot of records.
   - We could instead move the last record into the "hole" created by the deleted record (Figure 10.8).
   - This changes the order the records are in.
   - It turns out to be undesirable to move records to occupy freed space, as moving requires block accesses.
   - Also, insertions tend to be more frequent than deletions.
   - It is acceptable to leave the space open and wait for a subsequent insertion.
   - This leads to a need for additional structure in our file design.

3. So one solution is:
   - At the beginning of a file, allocate some bytes as a **file header**.
   - This header for now need only be used to store the address of the first record whose contents are deleted.

- This first record can then store the address of the second available record, and so on (Figure 10.9).
- To insert a **new** record, we use the record pointed to by the header, and change the header pointer to the **next** available record.
- If no deleted records exist we add our new record to the end of the file.

4. **Note**: Use of pointers requires careful programming. If a record pointed to is moved or deleted, and that pointer is not corrected, the pointer becomes a **dangling pointer**. Records pointed to are called **pinned**.
5. Fixed-length file insertions and deletions are relatively simple because "one size fits all". For variable length, this is not the case.

10.6.2 Variable-Length Records

1. Variable-length records arise in a database in several ways:
   - Storage of multiple items in a file.
   - Record types allowing variable field size
   - Record types allowing repeating fields
2. We'll look at several techniques, using one example with a variable-length record:

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

   Account-information is an array with an arbitrary number of elements.

Byte string representation

1. Attach a special end-of-record symbol ($\perp$) to the end of each record. Each record is stored as a string of successive bytes (See Figure 10.10).

   Byte string representation has several disadvantages:
   - It is not easy to re-use space left by a deleted record
   - In general, there is no space for records to grow longer. (Must move to expand, and record may be pinned.)

   So this method is not usually used.
2. An interesting structure: *Slot page structure*.

   There is a header at the beginning of each block, containing:
   - # of record entires in the header
   - the end of free space in the block
   - an array whose entries contain the location and size of each record.
3. The slot 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.

Fixed-length representation

1. Uses one or more fixed-length records to represent one variable-length record.
2. Two techniques:
   - **Reserved space** - uses fixed-length records large enough to accommodate the largest variable-length record. (Unused space filled with end-of-record symbol.)
   - **Pointers** - represent by a list of fixed-length records, chained together.
3. The reserved space method requires the selection of some maximum record length. (Figure 10.12)

   If most records are of near-maximum length this method is useful. Otherwise, space is wasted.
4. Then the pointer method may be used (Figure 10.13). Its disadvantage is that space is wasted in successive records in a chain as non-repeating fields are still present.
5. To overcome this last disadvantage we can split records into two blocks (See Figure 10.14)
   - **Anchor block** - contains first records of a chain
   - **Overflow block** - contains records other than first in the chain.

   Now all records in a block have the same length, and there is no wasted space.

10.7 Organization of Records in Files

There are several ways of organizing records in files.

- **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.
- **sequential file organization**. Records are stored in sequential order, based on the value of the search key of each record.
- **hashing file organization**. A hash function is computed on some attribute of each record. The result of the function specifies in which block of the file the record should be placed — to be discussed in chapter 11 since it is closely related to the indexing structure.
- **clustering file organization**. Records of several different relations can be stored in the same file. 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.

10.7.1 Sequential File Organization

1. A **sequential file** is designed for efficient processing of records in **sorted order** on some **search key**.
   - Records are chained together by pointers to permit fast retrieval in search key order.
   - Pointer points to next record in order.
   - Records are stored physically in search key order (or as close to this as possible).
   - This minimizes number of block accesses.
   - Figure 10.15 shows an example, with *bname* as the search key.
2. It is difficult to maintain physical sequential order as records are inserted and deleted.
   - Deletion can be managed with the pointer chains.
   - Insertion poses problems if no space where new record should go.

- If space, use it, else put new record in an **overflow block**.
- Adjust pointers accordingly.
- Figure 10.16 shows the previous example after an insertion.
- Problem: we now have some records out of physical sequential order.
- If very few records in overflow blocks, this will work well.
- If order is lost, reorganize the file.
- Reorganizations are expensive and done when system load is low.

3. If insertions rarely occur, we could keep the file in physically sorted order and reorganize when insertion occurs. In this case, the pointer fields are no longer required.

10.7.2 Clustering File Organization

1. One relation per file, with fixed-length record, is good for small databases, which also reduces the code size.
2. Many large-scale DB systems do not rely directly on the underlying operating system for file management. One large OS file is allocated to DB system and all relations are stored in one file.
3. To efficiently execute queries involving $depositor \bowtie customer$, one may store the *depositor* tuple for each *cname* near the customer tuple for the corresponding *cname*, as shown in Figure 10.19.
4. This structure mixes together tuples from two relations, but allows for efficient processing of the join.
5. If the customer has many accounts which cannot fit in one block, the remaining records appear on nearby blocks. This file structure, called *clustering*, allows us to read many of the required records using one block read.
6. Our use of clustering enhances the processing of a particular join but may result in slow processing of other types of queries, such as selection on customer.

   For example, the query

   select *
   from *customer*

   now requires more block accesses as our *customer* relation is now interspersed with the *deposit* relation.
7. Thus it is a trade-off, depending on the types of query that the database designer believes to be most frequent. Careful use of clustering may produce significant performance gain.

10.8 Data Dictionary Storage

1. The database also needs to store information **about** the relations, known as the **data dictionary**. This includes:

   - Names of relations.
   - Names of attributes of relations.
   - Domains and lengths of attributes.
   - Names and definitions of views.
   - Integrity constraints (e.g., key constraints).

   plus data on the system users:

   - Names of authorized users.
   - Accounting information about users.

plus (possibly) statistical and descriptive data:

- Number of tuples in each relation.
- Method of storage used for each relation (e.g., clustered or non-clustered).

2. When we look at indices (Chapter 11), we'll also see a need to store information about each index on each relation:
   - Name of the index.
   - Name of the relation being indexed.
   - Attributes the index is on.
   - Type of index.
3. This information is, in itself, a miniature database. We can use the database to store data about itself, simplifying the overall structure of the system, and allowing the full power of the database to be used to permit fast access to system data.
4. The exact choice of how to represent system data using relations must be made by the system designer. One possible representation follows.

   System-catalog-schema = (relation-name, number-attrs)
   Attr-schema = (attr-name, rel-name, domain-type, position, length)
   User-schema = (user-name, encrypted-password, group)
   Index-schema = (index-name, rel-name, index-type, index-attr)
   View-schema = (view-name, definition)

10.9 Storage Structures for Object-Oriented Databases (Omitted)

Interesting stuff, omitted due to lack of time.

10.9.1 Mapping of Objects to Files

10.9.2 Implementation of Object Identifiers

10.9.3 Management of Persistent Pointers: Pointer swizzling vs. hardware swizzling

10.9.4 Disk versus Memory Structure of Objects

10.9.5 Large Objects