# STORAGE AND FILE STRUCTURE

Engr. Laraib Siddiqui

## Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
  - data loss on power failure or system crash
  - physical failure of the storage device
- Can differentiate storage into:
  - **volatile storage:** loses contents when power is switched off
  - **non-volatile storage**:
    - Contents persist even when power is switched off.
    - Includes secondary and tertiary storage, as well as batter-backed up main-memory.

# Physical Storage Media

**Cache**

- fastest and most costly form of storage; volatile; managed by the computer system hardware.

**Main memory**

- fast access (generally too small (or too expensive) to store the entire database
- **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.

**Flash memory**

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again

**Magnetic-disk**

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- **direct-access** – possible to read data on disk in any order

**Optical storage**

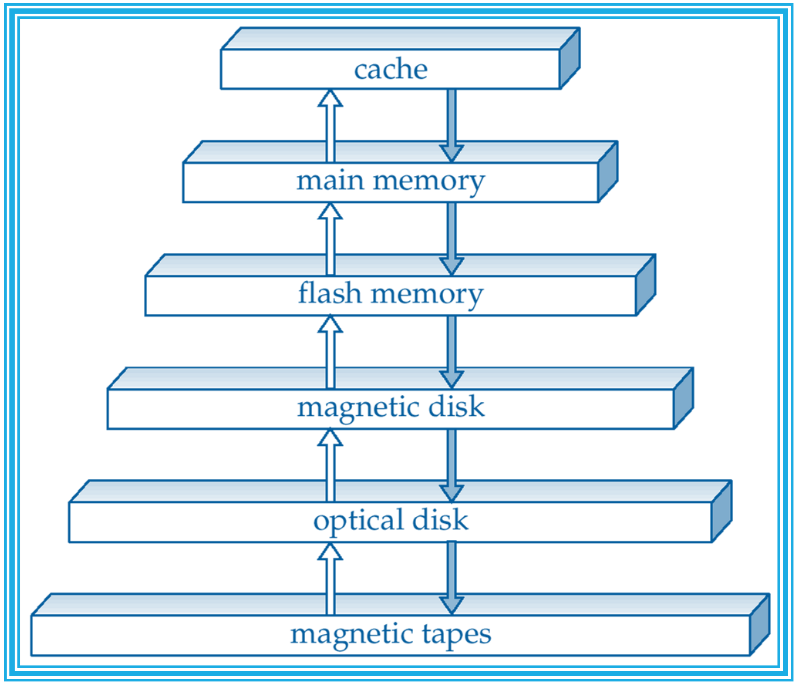
- non-volatile, data is read optically from a spinning disk using a laser
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms

**Tape storage**

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk

# Storage Hierarchy

- **Primary storage:** Fastest media but volatile (cache, main memory).
- **Secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **Tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage



## RAID

**RAID: Redundant Arrays of Independent Disks**

- disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
  - high capacity and high speed by using multiple disks in parallel, and
  - high reliability by storing data redundantly, so that data can be recovered even if a disk fails

**Redundancy** – store extra information that can be used to rebuild information lost in a disk failure

- E.g., **Mirroring** (or **shadowing**)
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other

## Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
  - Load balance multiple small accesses to increase throughput
  - Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
  - **Bit-level striping**
    - split the bits of each byte across multiple disks
  - **Block-level striping**
    - with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1

# RAID Levels

Schemes to provide redundancy at lower cost by using disk striping combined with p

**RAID Level 0**: Block striping; non-redundant.

Used in high-performance applications where data lost is not critical.

**RAID Level 1**: Mirrored disks with block striping

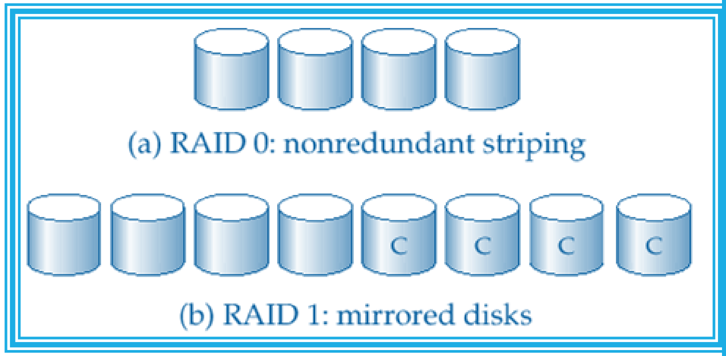
Offers best write performance.

Popular for applications such as storing log files in a database system.

**RAID Level 2**: Memory-Style Error-Correcting-Codes (ECC) with bit striping.

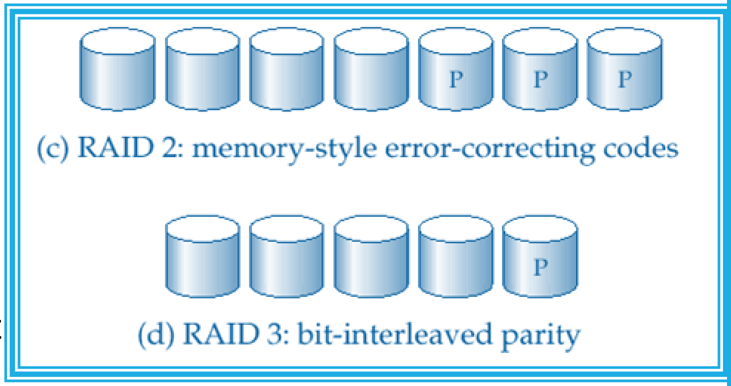
**RAID Level 3**: Bit-Interleaved Parity

a single parity bit is enough for error correction, not just detection, since we know which disk has failed parity bits



(a) RAID 0: nonredundant striping

(b) RAID 1: mirrored disks



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

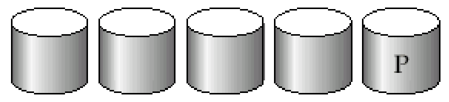
(d) RAID 3: bit-interleaved parity

## RAID Levels

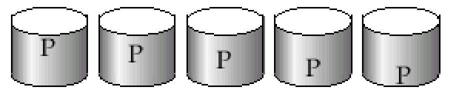
**RAID Level 4:** Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from *N* other disks.

**RAID Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk.

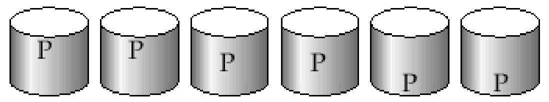
**RAID Level 6**: P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.



(e) RAID 4: block-interleaved parity



(f) RAID 5: block-interleaved distributed parity



(g) RAID 6: P + Q redundancy

## Storage Access

A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.

Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.

**Buffer** – portion of main memory available to store copies of disk blocks.

**Buffer manager** – subsystem responsible for allocating buffer space in main memory.

Programs call on the buffer manager when they need a block from disk.

- If the block is already in the buffer, the requesting program is given the address of the block in main memory
- If the block is not in the buffer,
  - the buffer manager allocates space in the buffer for the block, replacing (throwing out) some other block, if required, to make space for the new block.
  - The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
  - Once space is allocated in the buffer, the buffer manager reads the block from the disk to the buffer, and passes the address of the block in main memory to requester.

# Buffer-Replacement Policies

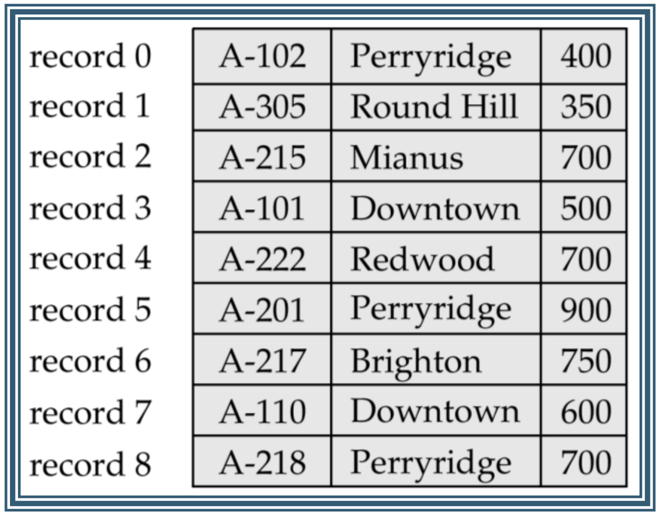
- Most operating systems replace the block **least recently used** (LRU strategy)
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer

## File Organization

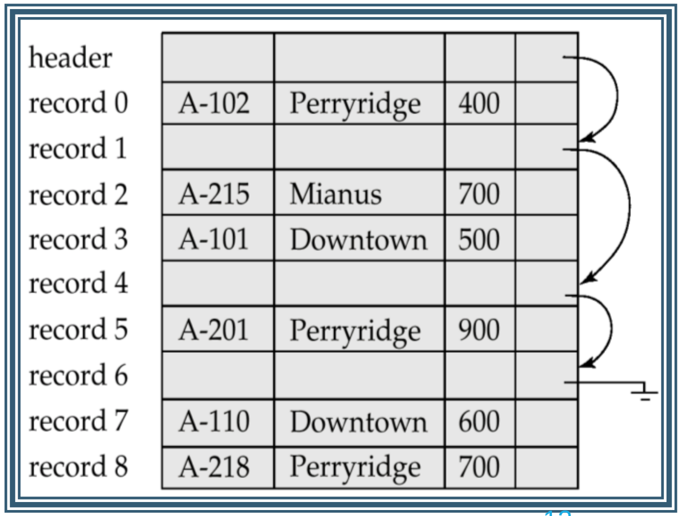
- The database is stored as a collection of *files*. Each file is a sequence of *records.* A record is a sequence of fields.
- One approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

# Fixed-Length Records

- Simple approach:
  - Store record $i$ starting from byte $n * (i - 1)$, where $n$ is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries
- Deletion of record $I$: alternatives:
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record $n$ to $i$
  - do not move records, but link all free records on a *free list*
- **Free Lists**
- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)



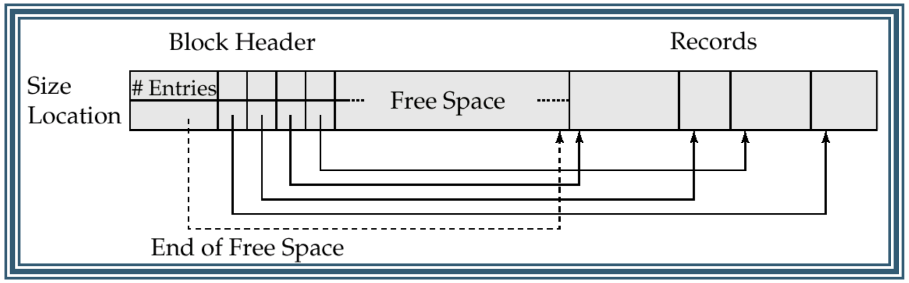
| | | | |
|---|---|---|---|
| record 0 | A-102 | Perryridge | 400 |
| record 1 | A-305 | Round Hill | 350 |
| record 2 | A-215 | Mianus | 700 |
| record 3 | A-101 | Downtown | 500 |
| record 4 | A-222 | Redwood | 700 |
| record 5 | A-201 | Perryridge | 900 |
| record 6 | A-217 | Brighton | 750 |
| record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



| | | | | |
|---|---|---|---|---|
| header | | | | |
| record 0 | A-102 | Perryridge | 400 | |
| record 1 | | | | |
| record 2 | A-215 | Mianus | 700 | |
| record 3 | A-101 | Downtown | 500 | |
| record 4 | | | | |
| record 5 | A-201 | Perryridge | 900 | |
| record 6 | | | | |
| record 7 | A-110 | Downtown | 600 | |
| record 8 | A-218 | Perryridge | 700 | |

# 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 (used in some older data models).
- Byte string representation
  - Attach an *end-of-record* (⊥) control character to the end of each record
  - Difficulty with deletion
  - Difficulty with growth

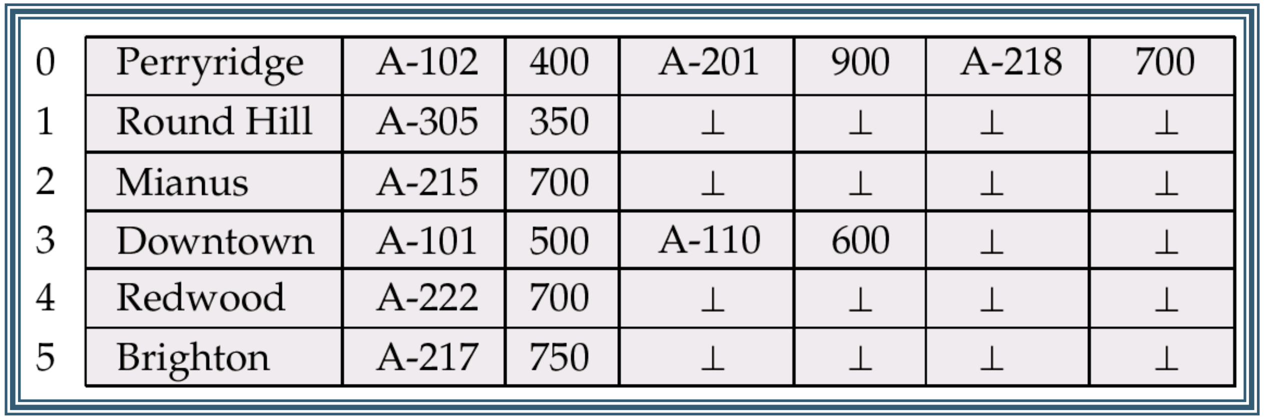


**Slotted Page Structure**

- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.

## Variable-Length Records (Cont.)

- Fixed-length representation:
  - reserved space
  - pointers
- Reserved space – can use fixed-length records of a known maximum length; unused space in shorter records filled with a null or end-of-record symbol.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 | ⊥ | ⊥ | ⊥ | ⊥ |

## Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space. Heap File does not support any ordering, sequencing, or indexing on its own.
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- **Clustering -** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

## Data Dictionary Storage

Data dictionary (also called system catalog) stores metadata: that is, data about data, such as

- Information about relations
  - names of relations
  - names and types of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/…)
  - Physical location of relation
    - operating system file name or
    - disk addresses of blocks containing records of the relation
- Information about indices