Part 12:
Hashing and RAID

Database System Concepts, 7th Ed.

Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block)
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *A*; i.e. from the key space to the address space

$$h : K \rightarrow A$$

- Hash function is used to locate entries for access, insertion as well as deletion
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

Hash Functions

- Worst hash function maps all search-key values to the same bucket
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of all possible values
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file
- Typical hash functions perform computation on the internal binary representation of the search-key
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key

- There are 10 buckets
- The binary representation of the *i*th character is treated as an integer
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g., h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3

Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

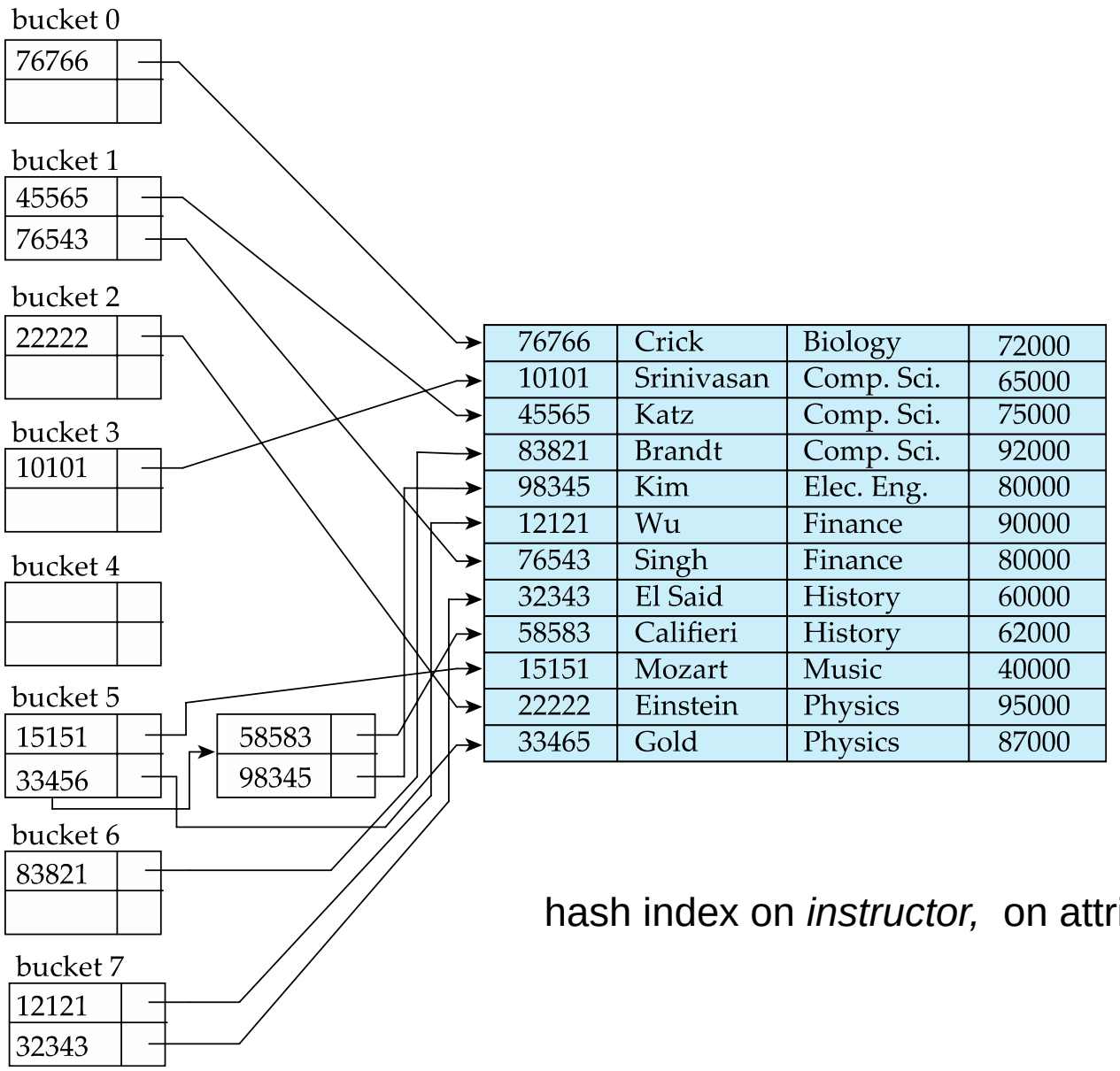
bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key.

Example of Hash Index



hash index on *instructor,* on attribute *ID*

Handling of Bucket Overflows

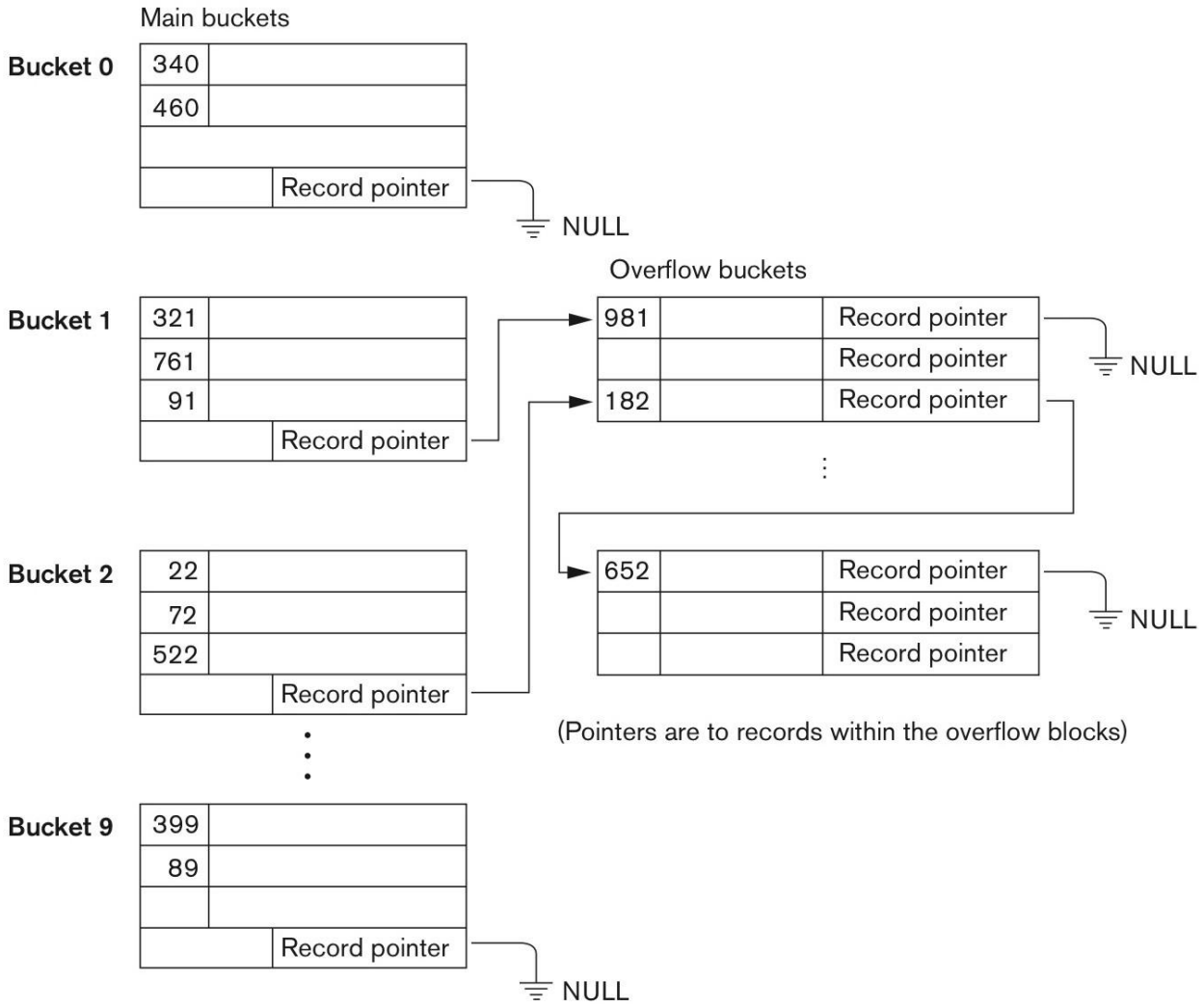
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records, which can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***
- Suppose there are *n* records and *r* buckets, each with capacity *C,* then assuming that each bucket is chosen with the same probability 1/*r* , then for a *particular* bucket, the probability of overflow is

$$P = 1 - \sum_{k=0}^{C} \binom{n}{k} \left(\frac{1}{r}\right)^k \left(1 - \frac{1}{r}\right)^{n-k}$$

Handling of Overflows

- **Collision**
  - Hash field value for inserted record hashes to an address already occupied
- **Collision Resolution**
  - **Open Addressing** – proceed from the occupied position and find the next unused position and place record there
  - **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list

Overflow Chaining



Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of bucket addresses, but databases will grow or shrink with time
  - If the initial number of buckets is too small, and the file grows, performance will degrade due to too much overflows
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be under full).
  - If database shrinks, again space will be wasted
- Solution: allow the number of buckets to change dynamically

Hashing with Dynamic File Expansion

- These hashing schemes take advantage of the fact that the result of applying a hashing function is a nonnegative integer and hence can be represented as a binary number
- The access structure is built on the binary representation of the hashing function result, which is a string of bits called the **hash value** of a record
- Records are distributed among buckets based on the values of the leading bits in their hash values
- We shall look at two such schemes: **extendible hashing** and **dynamic hashing**

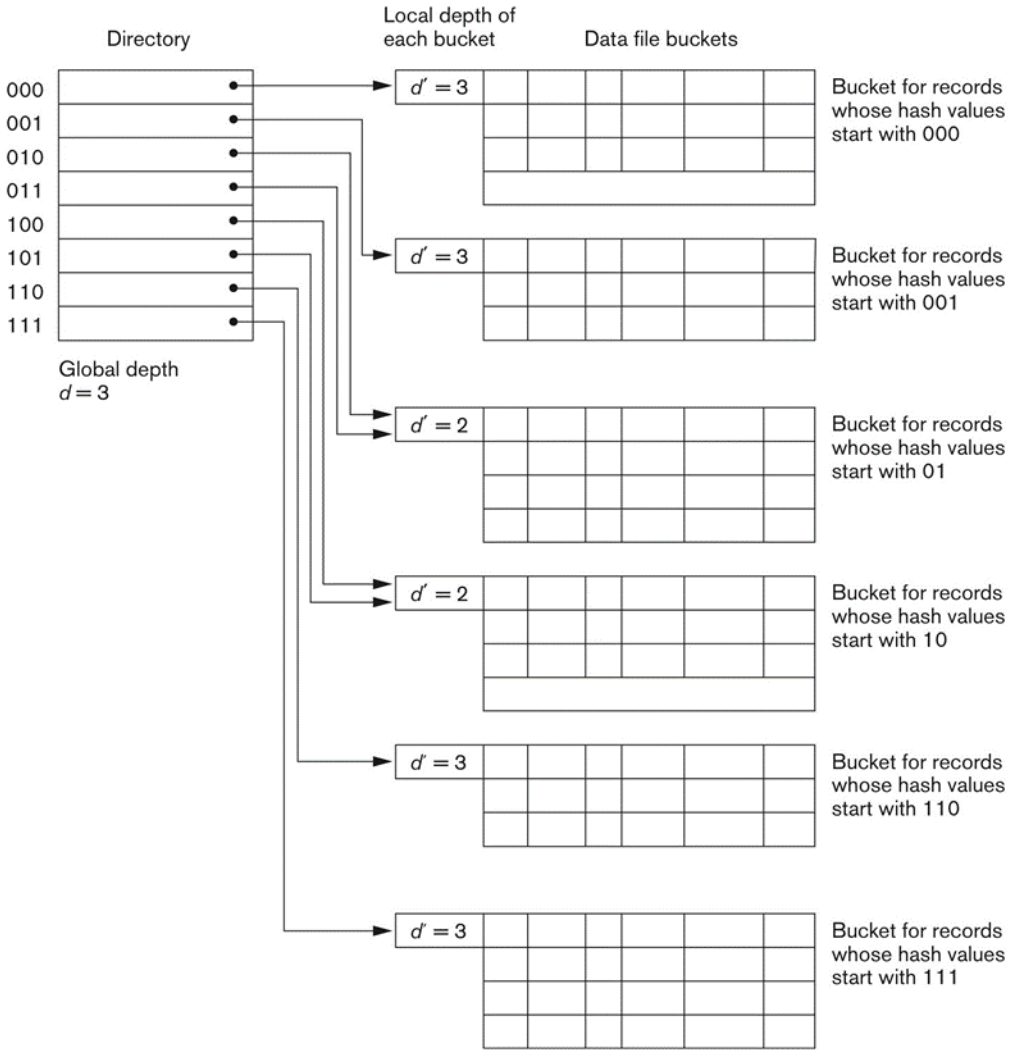
Extendible Hashing

- A directory — an array of 2^d bucket addresses — is maintained, where *d* is called the **global depth** of the directory
- The integer value corresponding to the first (high-order) *d* bits of a hash value is used as an index to the array to determine a directory entry, and the address in that entry determines the bucket in which the corresponding records are stored
  - there does not have to be a distinct bucket for each of the 2^d directory locations
- Several directory locations with the same first *d′* bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket

Extendible Hashing

- A **local depth** *d′* — stored with each bucket — specifies the number of bits on which the bucket contents are based
- The value of *d* can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array
- Doubling is needed if a bucket, whose local depth *d′* is equal to the global depth *d*, overflows
- Halving occurs if *d* > *d′* for all the buckets after some deletions occur
- Most record retrievals require two block accesses — one to the directory and the other to the bucket

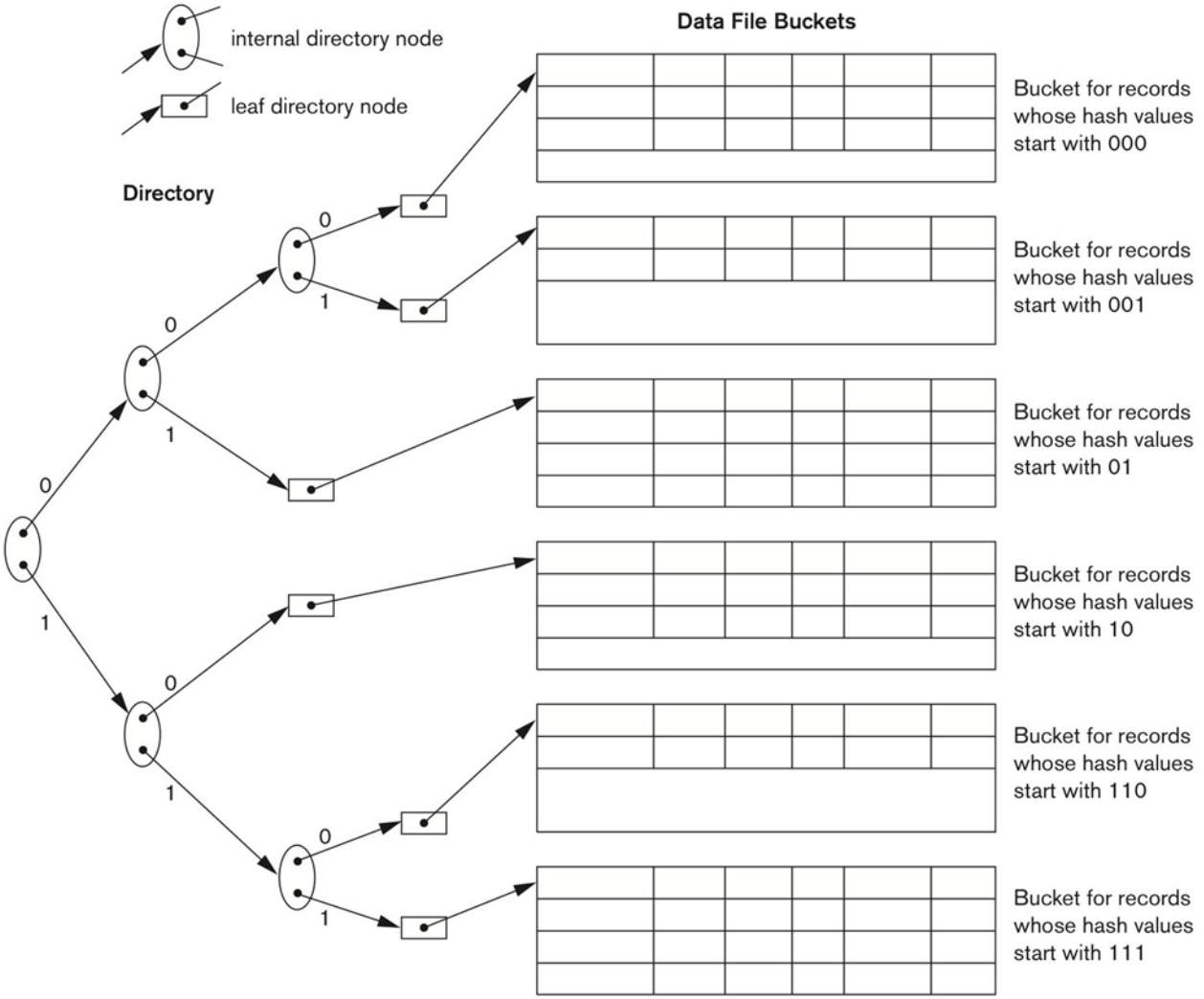
Extendible Hashing



Dynamic Hashing

- The eventual storage of records in buckets for dynamic hashing is somewhat similar to extendible hashing — the major difference is in the organization of the directory
- Whereas extendible hashing uses the notion of global depth (high-order *d* bits) for the flat directory and then combines adjacent collapsible buckets into a bucket of local depth $d - 1$, dynamic hashing maintains a tree-structured directory with two types of nodes:
  - Internal nodes that have two pointers—the left pointer corresponding to the 0 bit (in the hashed address) and a right pointer corresponding to the 1 bit.
  - Leaf-nodes—these hold a pointer to the actual bucket with records

Dynamic Hashing



RAID

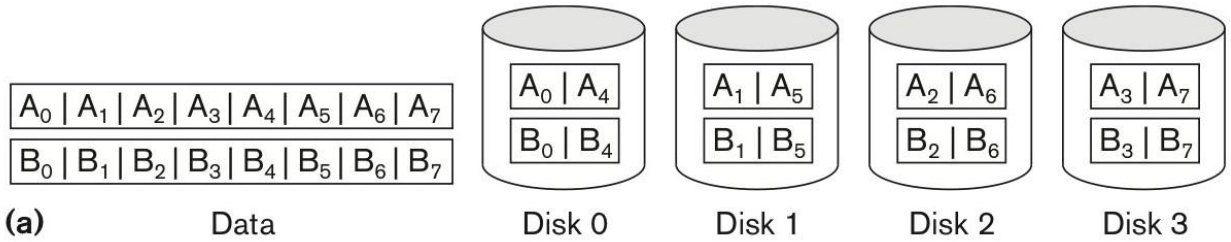
- **RAID: Redundant Arrays of Independent Disks**
  - disk organization techniques that manage a large numbers of disks, providing
    - **high capacity** and **high speed** by using multiple disks in parallel,
    - **high reliability** by storing data redundantly, so that data can be recovered even if a disk fails
- 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
  - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx.11 years), will have a system MTTF of 1000 hours (approx. 41 days)
    - Failure rate of one disk r = 1/MTTF. Aggregate failure rate of 100 disks = 100r = 100/MTTF, giving a system MTTF of MTTF/100
  - Techniques for using redundancy to avoid data loss are critical with a large numbers of disks

Improvement of Reliability via Redundancy

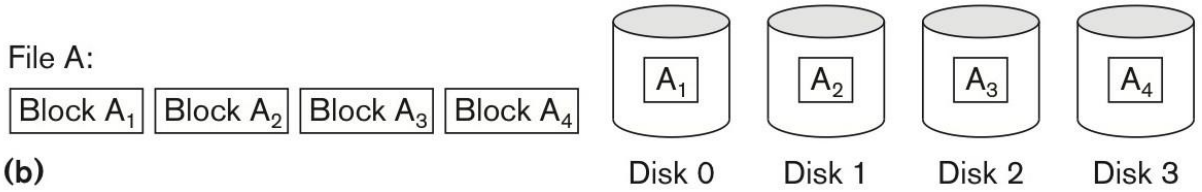
- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
  - Duplicate every disk. Logical disk consists of two physical disks
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
    - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
      - Probability of combined event is very small
        - Except for dependent failure modes such as fire or building collapse or electrical power surges

Improvement in Performance via Parallelism

- Improve transfer rate by striping data across multiple disks
- **Bit-level striping** – split the bits of each byte across multiple disks
- **Block-level striping** – splits the blocks of a file across multiple disks
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel



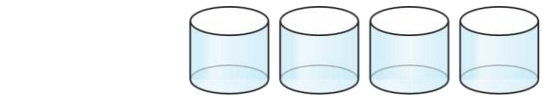
Bit-level striping



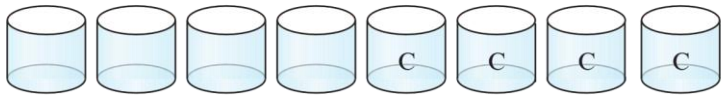
Block-level striping

RAID Levels

- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
  - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **RAID Level 0**: Block striping; non-redundant
  - Used in high-performance applications where data loss is not critical
- **RAID Level 1**: Mirrored disks with or without block striping
  - with block striping it is often called **Level 1+0 or 10**



(a) RAID 0: nonredundant striping



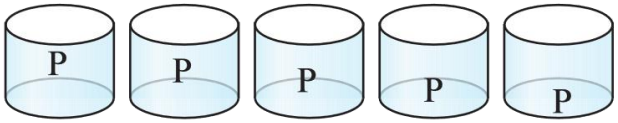
(b) RAID 1: mirrored disks

RAID Levels

- **Parity blocks**: Parity block *j* stores XOR of bits from block *j* of each disk
  - The XOR'ed parity bit corresponds to the vector addition of the constituent bits and results in overall even parity
    - the XOR bit for more than 2 bit patterns: just calculate number of 1's in the corresponding bits. If it is even or zero then that XOR'ed bit is 0. If it is odd then that XOR'ed bit is 1.
  - To recover data for a block, set the missing bit such that even parity results

RAID Levels

- **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
  - Block writes occur in parallel if the blocks and their parity blocks are on different disks



(c) RAID 5: block-interleaved distributed parity

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

RAID Levels

- **RAID Level 4**
  - Like RAID 5 but all parity blocks are stored on one disk, the **parity disk**
  - Every write will involve the parity disk and it can become a bottleneck
  - Normal reads will not use the parity disk at all
- **RAID Level 6**: P+Q Redundancy scheme; similar to Level 5, but stores two error correction blocks (P, Q) instead of single parity block to guard against multiple disk failures
  - Better reliability than Level 5 but at a higher cost