Data Storage - II:

Efficient Usage & Errors

Contains slides from:
Naci Akkök, Hector Garcia-Molina, Pål Halvorsen, Ketil Lund

Overview

- ✓ Efficient storage usage
- ✓ Disk errors
- ✓ Error recovery

Efficient Storage Usage

Efficient Secondary Storage Usage

- ✓ Many programs are assumed to fit in main memory, but when implementing a DBS one must assume that data is larger than main memory

- ✓ Must take into account the use of secondary storage
  - ➢ there are large access time gaps, i.e., a disk access will probably dominate the total execution time
  - ➢ there may be huge performance improvements if we reduce the number of disk accesses
  - ➢ a “slow” algorithm with few disk accesses will probably outperform a “fast” algorithm with many disk accesses

- ✓ Several ways to optimize .....

Block Size – I

- ✓ The block size may have large effects on performance
- ✓ Example:

  assume random block placement on disk and sequential file access
  - ➢ doubling block size will halve the number of disk accesses
    - each access take some more time to transfer the data, but the total time is the same (i.e., more data per request)
    - halve the seek times
    - halve rotational delays are omitted
  - ➢ e.g., when increasing block size from 2 KB to 4 KB (no gaps,...) for *cheetah X15* typically an average of:
    - ☺ 3.6 ms is *saved* for seek time
    - ☺ 2 ms is *saved* in rotational delays
    - ☹ 0.026 ms is *added* per transfer time

    } saving a total of 5.6 ms when reading 4 KB (49,8 %)
  - ➢ e.g., increasing from 2 KB to 64 KB saves ~96,4 % reading 64 KB

Block Size – II

- ✓ Thus, increasing block size can increase performance by reducing seek times and rotational delays

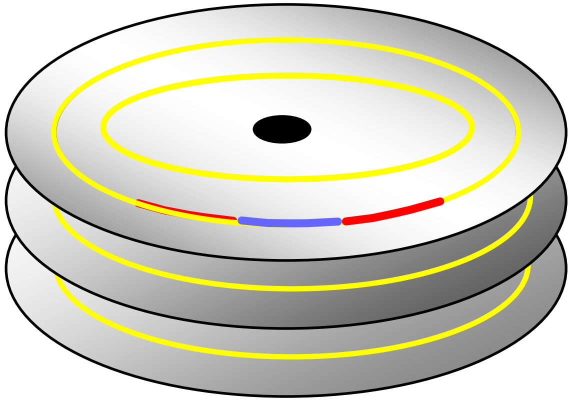
- ✓ However, a large block size is not always best
  - ➢ blocks spanning several tracks still introduce latencies
  - ➢ small data elements may occupy only a fraction of the block

- ✓ Which block size to use therefore depend on data size and data reference patterns

- ✓ The trend, however, is to use large block sizes as new technology appear with increased performance

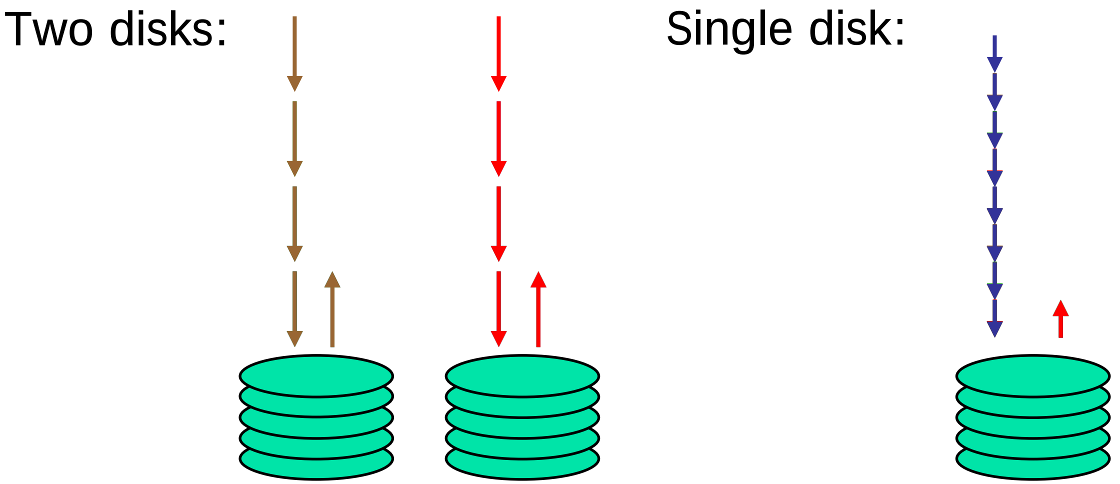
Using Adjacent Sectors, Cylinders and Tracks

- ✓ To avoid seek time (and possibly rotational delay), we can *store data likely to be accessed together* on
  - ➢ adjacent sectors
    (similar to using larger blocks)
  - ➢ if the track is full, use another track on the same cylinder
    (only use another head)
  - ➢ if the cylinder is full, use next cylinder
    (track-to-track seek)
- ✓ Advantage
  - ➢ can approach theoretical transfer rate (no seeks or rotational delays)
- ✓ Disadvantage
  - ➢ no gain if we have unpredictable disk accesses



Multiple Disks

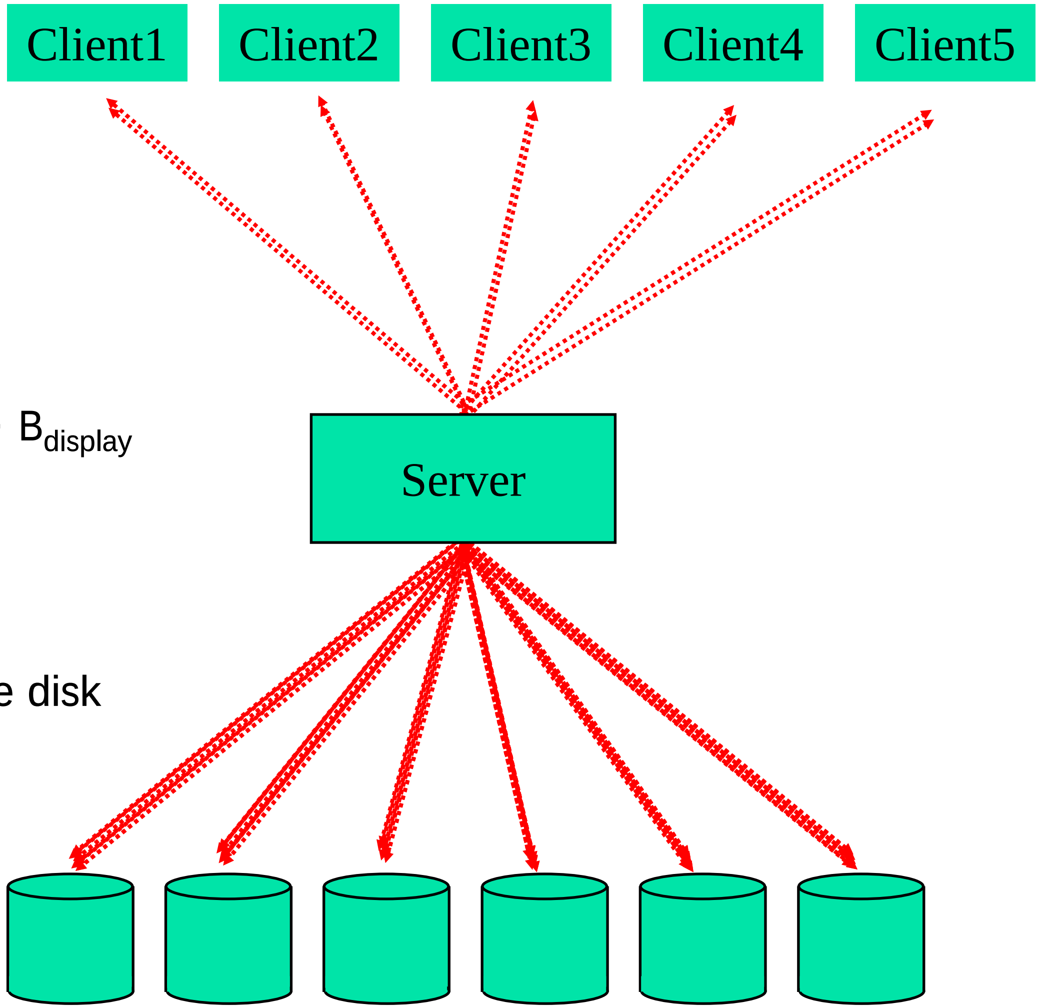
- ✓ Disk controllers and busses manage several devices
- ✓ One *can* improve total system performance by replacing one large disk with many small accessed in parallel
- ✓ Several independent heads can read simultaneously (if the other parts of the system can manage the speed)



Note 1:
the single disk might be faster, but as seek time and rotational delay are the dominant factors of total disk access time, the two smaller disks might operate faster together...

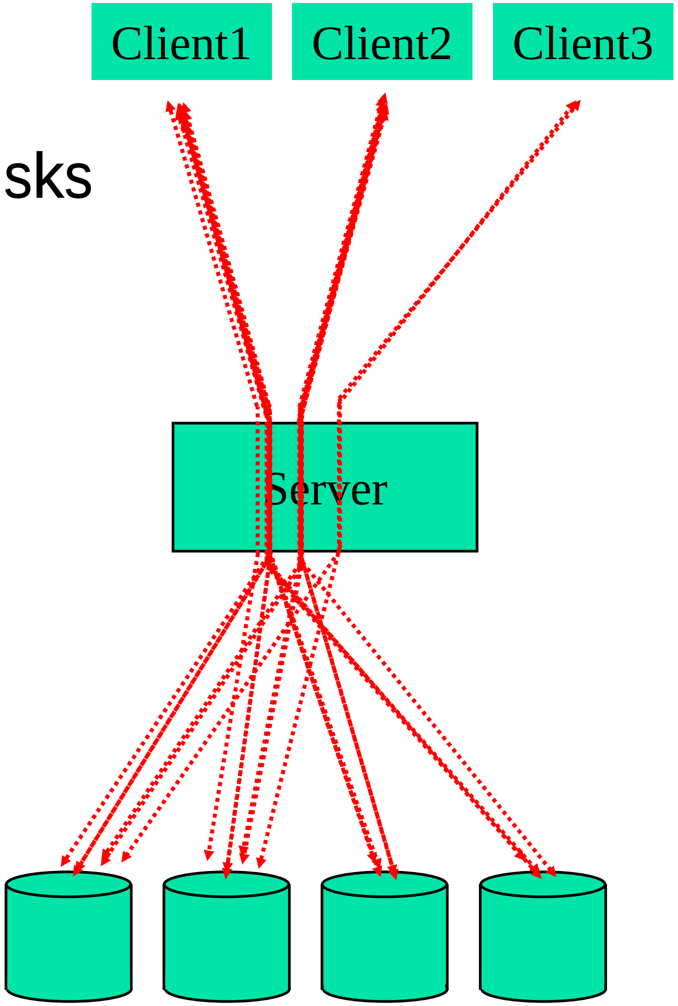
Multiple Disks: Striping

- ✓ Another reason to use multiple disks is when one disk cannot deliver requested data rate
- ✓ In such a scenario, one might use several disks for striping:
  - bandwidth disk: B_{disk}
  - required bandwidth: $B_{display}$
  - $B_{display} > B_{disk}$
  - read from *n* disks in parallel: $n\,B_{disk} > B_{display}$
  - clients are serviced in *rounds*
- ✓ Advantages
  - high data rates
  - faster response time compared to one disk
- ✓ Drawbacks
  - can't serve multiple clients in parallel
  - positioning time increases (i.e., reduced efficiency)



Multiple Disk: Interleaving

- ✓ Full striping usually not necessary today:
  - faster disks
  - better compression algorithms
- ✓ Interleaving lets each client be serviced by only a set of the available disks
  - make groups
  - ”stripe” data in a way such that
    a consecutive request arrive at
    next group (here each disk is a group)
- ✓ Advantages
  - multiple clients can still be served in parallel
  - more efficient disks
  - potentially shorter response time
- ✓ Drawbacks
  - load balancing
    (all clients access same group)

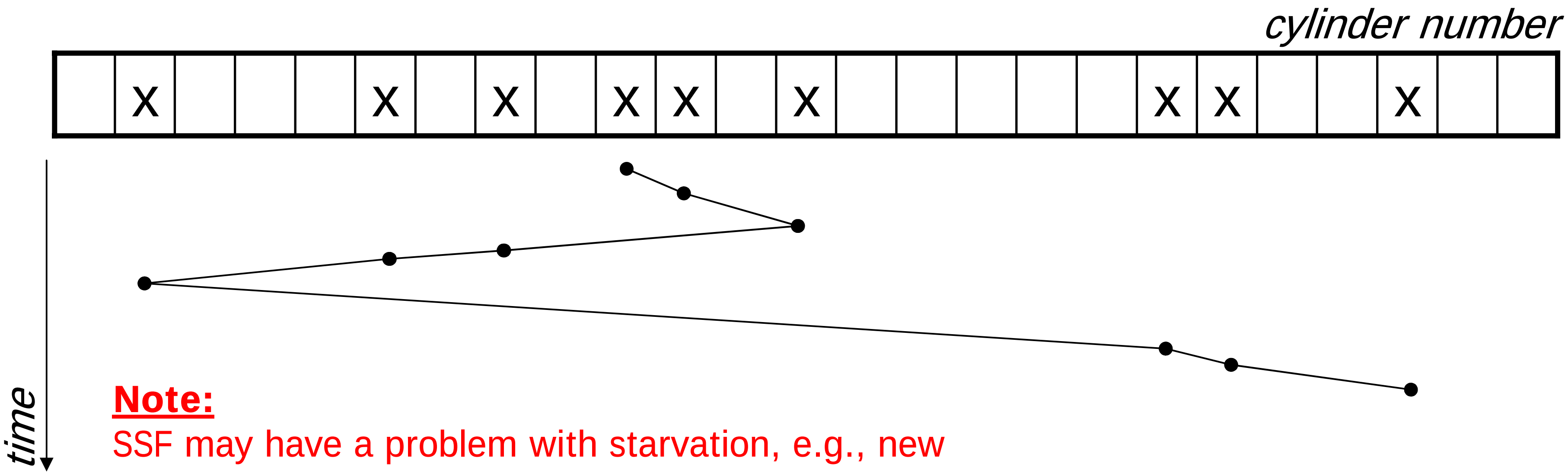


Multiple Disks: Mirroring

- ✓ Multiple disks might come in the situation where all requests are for one of the disks and the rest lie idle
- ✓ In such cases, it might make sense to have replicas of the data on several disks – if we have identical disks, it is called mirroring
- ✓ Advantages
  - ➢ faster response time
  - ➢ survive crashes – fault tolerance
  - ➢ load balancing by dividing the requests for the data on the same disks equally among the mirrored disks
- ✓ Drawbacks
  - ➢ increases storage requirement

Disk Scheduling – I

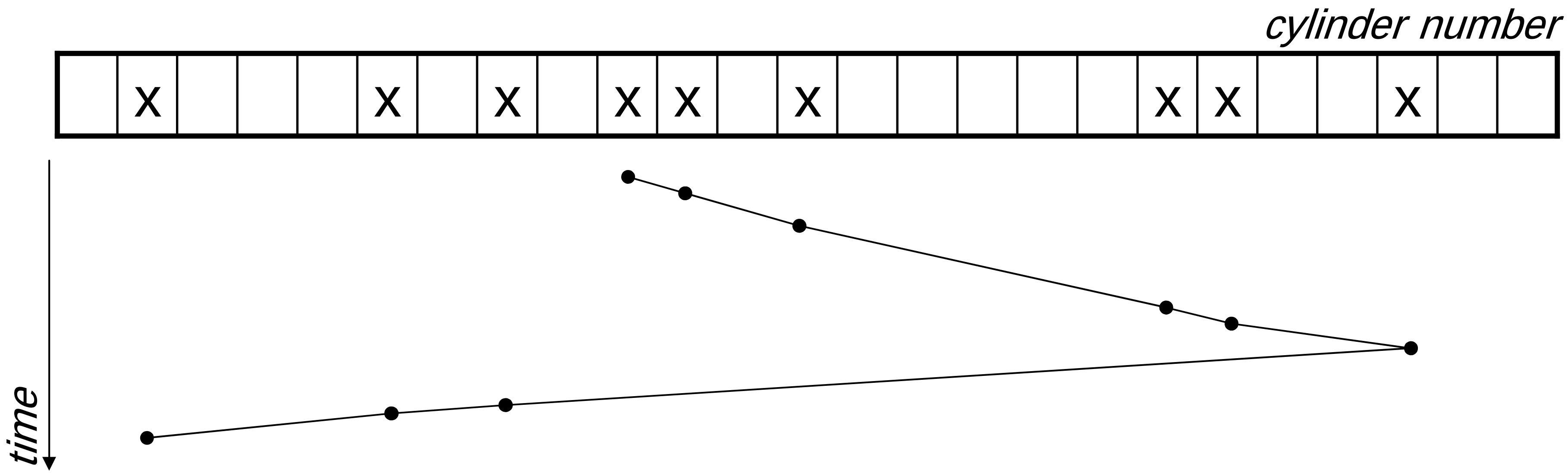
- Seek time is a dominant factor of total disk I/O time
- Let disk controller choose which request to serve next depending on current position on disk and requested block's position on disk (disk scheduling)
- Several algorithms
  - First-Come-First-Serve
  - Shortest Seek First (SSF): serve the request closest to current position



Note:
SSF may have a problem with starvation, e.g., new requests keep arriving whose positions are close to the current position – blocks far away will never be read

Disk Scheduling – II

- Elevator (SCAN) algorithm: make head do sweeps from innermost to outermost cylinder, make a stop if passing over a requested block, reverse direction if there are no more requests in the current direction



- Several other algorithms.....

Prefetching / Multiple Buffering – I

- ✓ If we can predict the access pattern, one might speed up performance using prefetching
  - ➢ eases disk scheduling
  - ➢ read larger amounts of data per request
  - ➢ data in memory when requested – reducing page faults

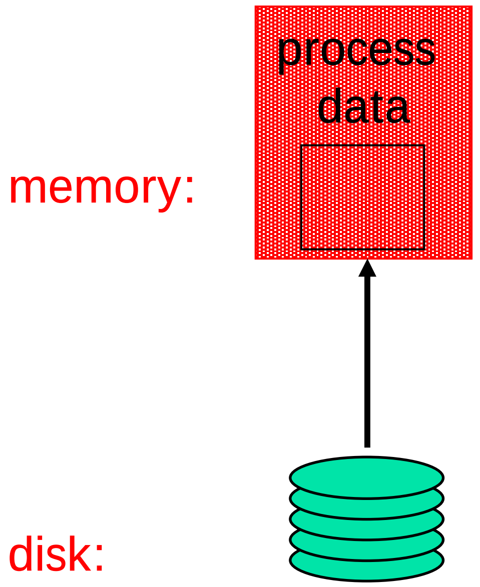
- ✓ One way of doing prefetching is double (multiple) buffering:
  - ➢ read data into first buffer
  - ➢ process data in *first* buffer and at the same time read data into *second* buffer
  - ➢ process data in *second* buffer and at the same time read data into *first* buffer
  - ➢ etc.

Prefetching / Multiple Buffering – II

- ✓ Example:
  have a file with block sequence B1, B2, ...
  our program processes data sequentially, i.e., B1, B2, ...

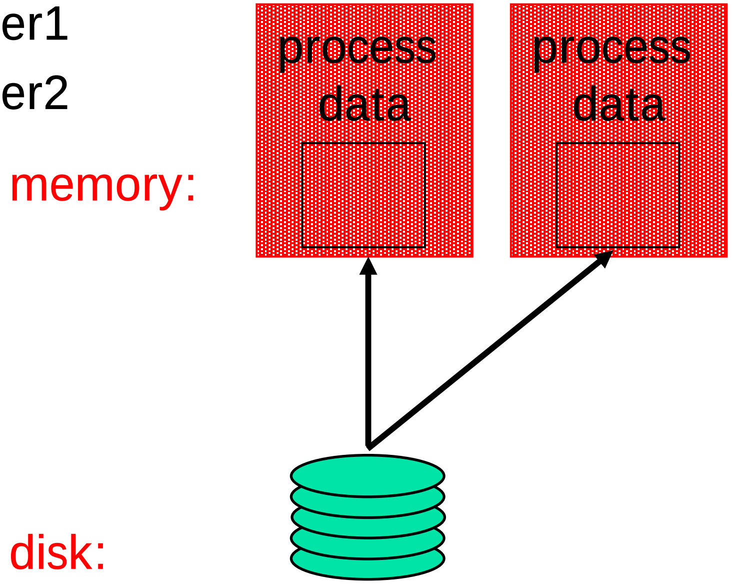
  - ➢ single buffer solution:
    - read B1 → buffer
    - process data in buffer
    - read B2 → buffer
    - process data in Buffer
    - ...
    - if P = time to process/block
      R = time to read in 1 block
      n = # blocks

single buffer time = n (P+R)



Prefetching / Multiple Buffering – III

- double buffer solution:
  - read B1 → buffer1
  - process data in buffer1, read B2 → buffer2
  - process data in buffer2, read B3 → buffer1
  - process data in buffer1, read B4 → buffer2
  - ...



  - if P = time to process/block
    R = time to read in 1 block
    n = # blocks

    if $P \geq R$
    double buffer time = $R + nP$

- if $P < R$, we can try to add buffers (*n* - buffering)

Disk Errors

Disk Errors – I

- ✓ Disk errors are rare:

	Barracuda 180	*Cheetah 36*	*Cheetah X15*
mean time to failure (MTTF)	1.2×10^6	1.2×10^6	36.7
recoverable errors	10 per 10^{12}	10 per 10^{12}	10 per 10^{12}
unrecoverable errors	1 per 10^{15}	1 per 10^{15}	1 per 10^{15}
seek errors	10 per 10^8	10 per 10^8	10 per 10^8

Note 1:
MTTF is the time in hours between each time the disk crashes

Note 2:
how often do we read wrong values – corrected when re-reading

Note 3:
how often do we get permanent errors on a sector – data moved to spare tracks

Note 4:
how often do we move the arm wrong (over wrong cylinder) – make another

Disk Errors – II

- ✓ Nevertheless, a disk can fail in several ways
  - intermittent failure –
    temporarily errors corrected by re-reading the block, e.g., dust on the patter making a bit value wrong
  - media decay/write errors –
    permanent errors where the bits are corrupted, e.g., disk head touches the platter and damages the magnetic surface
  - disk crashes –
    the entire disk becomes permanent unreadable

Checksums – I

- ✓ Disk sectors are stored with some redundant bits, called *checksums*
- ✓ Used to validate a read or written sector:
  - ➢ read sector and stored checksum
  - ➢ compute checksum on read sector
  - ➢ compare read and computed checksum

- ✓ If the validation fails (read and computed checksum differ), the read operation is repeated until
  - ➢ the read operation succeed → return correct content
  - ➢ the limit of retries is reached → return error “bad disk block”

Checksums – II

- ✓ Many different ways to compute checksums
  - ➢ 1-bit parity: count 1’s in block
    - even number: parity bit 0
    - odd number: parity bit 1
    - large chance of not detecting errors
  - ➢ Use more redundant bits
    - 8-bit parity: one parity bit per bit in a byte (count 1’s in most significant bit, ….) → decrease amount of missed errors
    - n-bit parity: chance of missing an error is $1/2^n$
  - ➢ Polynomial codes – CRC (cyclic redundancy check) :
    - following properties of binary numbers if using modulo-2 arithmetic
    - generate a single set of check digits based on the code
  - ➢ Reed-Solomon, 1-complement sum, ….
- ✓ Checksums only *detect* errors

Stable Storage

- ✓ The stable storage policy may solve some errors
  - ➢ applicable on one or more disks
  - ➢ use checksums for temporarily read errors
  - ➢ sectors are paired to represent one sector X
  - ➢ X is represented by X_L and X_R
  - ➢ writing policy:
    1. write X into X_L - use checksum to validate, if wrong retry. If still wrong, assume a media failure and use spare sector for X_L
    2. repeat (1) for X_R
  - ➢ read policy
    1. read X_L – use checksum to validate. If OK, return X_L as X. If wrong retry.
    2. if X_L cannot be read, repeat (1) for X_R
- ✓ Stable storage *doubles storage requirement*, but can for example correct some media failures
  (if either X_L or X_R correct, X can always be read)

Error Recovery

Crash Recovery

- ✓ The most serious type of errors are disk crashes, e.g.,
  - ➢ head have touched platter and is damaged
  - ➢ platters are out of position
  - ➢ ...
- ✓ No way to restore data unless we have a backup on another medium, e.g., tape, mirrored disk, etc.
- ✓ A number of schemes have been developed to reduce the probability of data loss during permanent disk errors
  - ➢ usually using an extended parity check
  - ➢ most known are the Redundant Array of Independent Disks (RAID) strategies

Disk Failure Models

- ✓ Our Seagate disks have a mean-time-to-failure of 55 years (at this time ~50 % of the disks are damaged), but
  - ➢ many disks fail during the first months (production errors)
  - ➢ if no production errors, disks will probably work many years
  - ➢ old disks have again a larger probability of failure due to accumulated effects of dust, etc.

Modulo-2 Sum – I

- ✓ Many parity schemes use *modulo-2 sum*, or also called *exclusive OR (XOR)*, to generate a redundant correction block
- ✓ The modulo-2 sum is performed by letting the i-th bit of the sum to be
  - ➢ 1 – if an odd number of blocks have 1 in the i-th position
  - ➢ 0 - if an even number of blocks have 1 in the i-th position
- ✓ Example

block 1	1	1	1	1	0	0	0	0
block 2	1	0	1	0	1	0	1	0
block 3	0	0	1	1	1	0	0	0
modulo-2 sum	0	1	1	0	0	0	1	0

Modulo-2 Sum – II

✓ Let $\oplus$ be the modulo-2 sum operator. Then ...

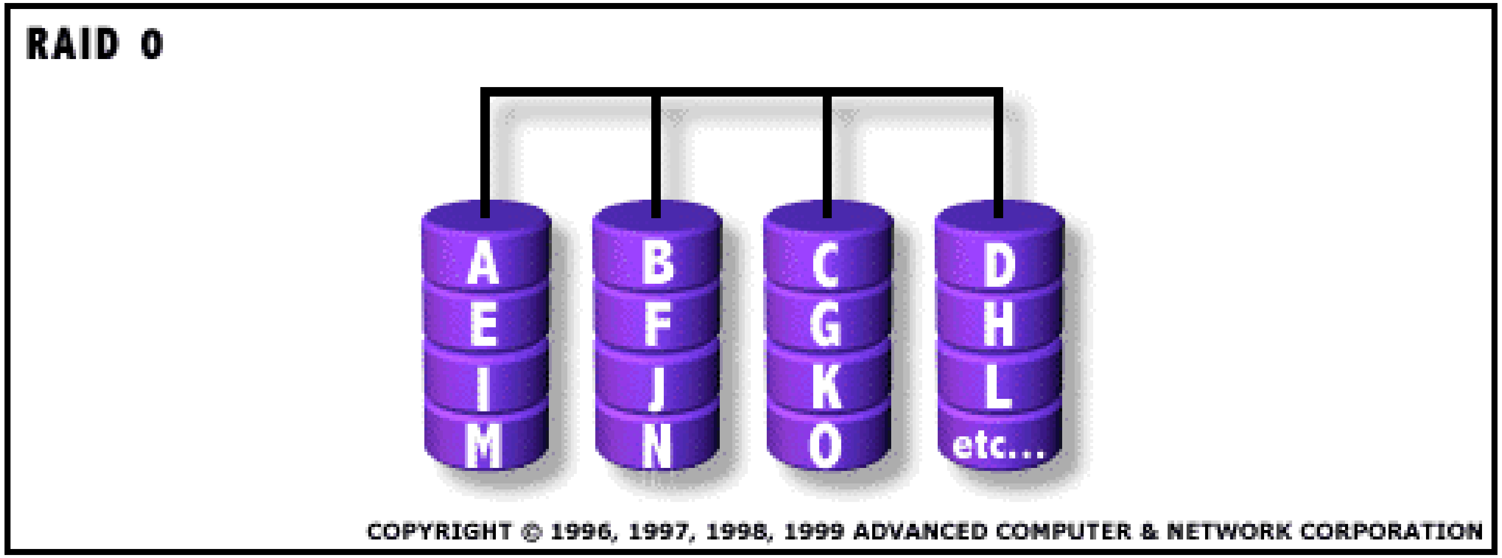
- ... the *commutative* law says that
  $x \oplus y = y \oplus x$
- ... the *associative* law says that
  $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
- ... the *identity* is 0, i.e.,
  $x = 0 \oplus x = x \oplus 0$
- ... $\oplus$ is its own inverse, i.e.,
  $x \oplus x = 0$

RAID (Redundant Array of Inexpensive Disks)

- ✓ RAID level 0: non-redundant
- ✓ RAID level 1: mirrored
- ✓ RAID level 2: memory-style error correcting code (ECC)
- ✓ RAID level 3: bit-interleaved parity
- ✓ RAID level 4: block-interleaved parity
- ✓ RAID level 5: block-interleaved distributed-parity
- ✓ RAID level 6: P+Q redundancy

RAID 0 (non-redundant) – I

- ✓ RAID 0: striped disk array without fault tolerance
- ✓ Data is broken down into blocks and each block is written to a separate disk



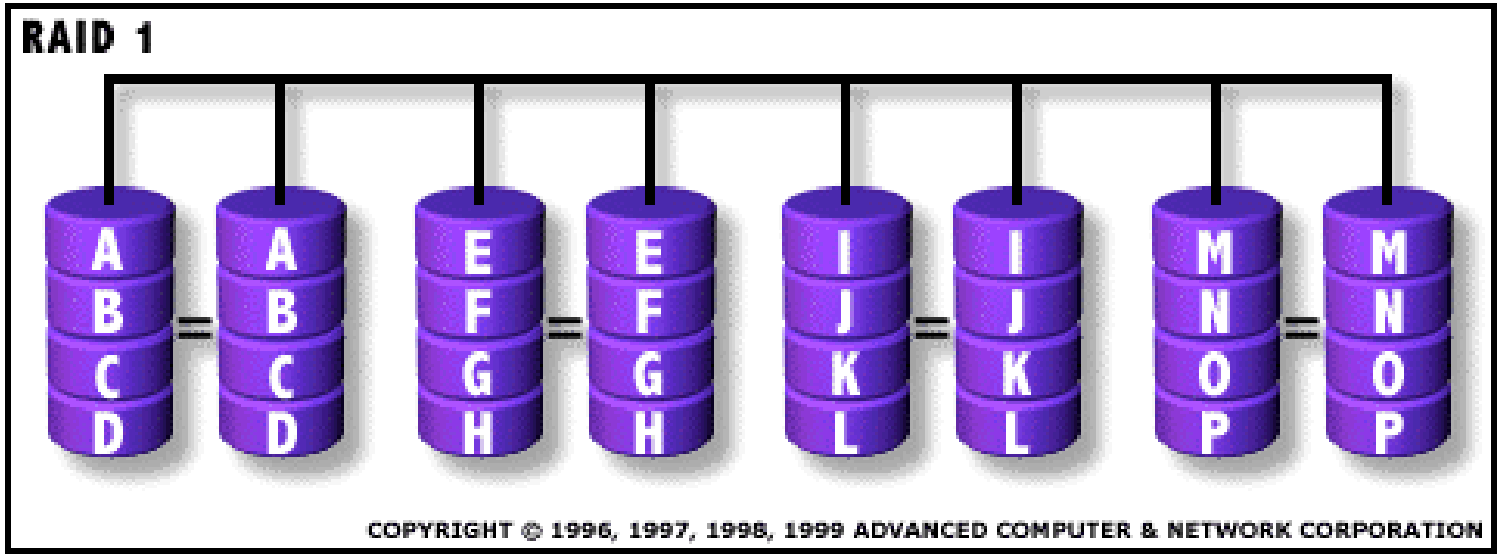
RAID 0 (non-redundant) – II

- Advantages
  - I/O performance is greatly improved by spreading the I/O load across many channels and drives
  - best performance is achieved when data is striped across multiple controllers with only one drive per controller (remember that the performance of one disk has improved)

- Disadvantages
  - not a "True" RAID because it is NOT fault-tolerant
  - the failure of just one drive will result in all data in an array being lost
  - should never be used in error-critical environments

RAID 1 (mirroring) – I

- ✓ RAID 1: mirroring
- ✓ Data is duplicated on another disk

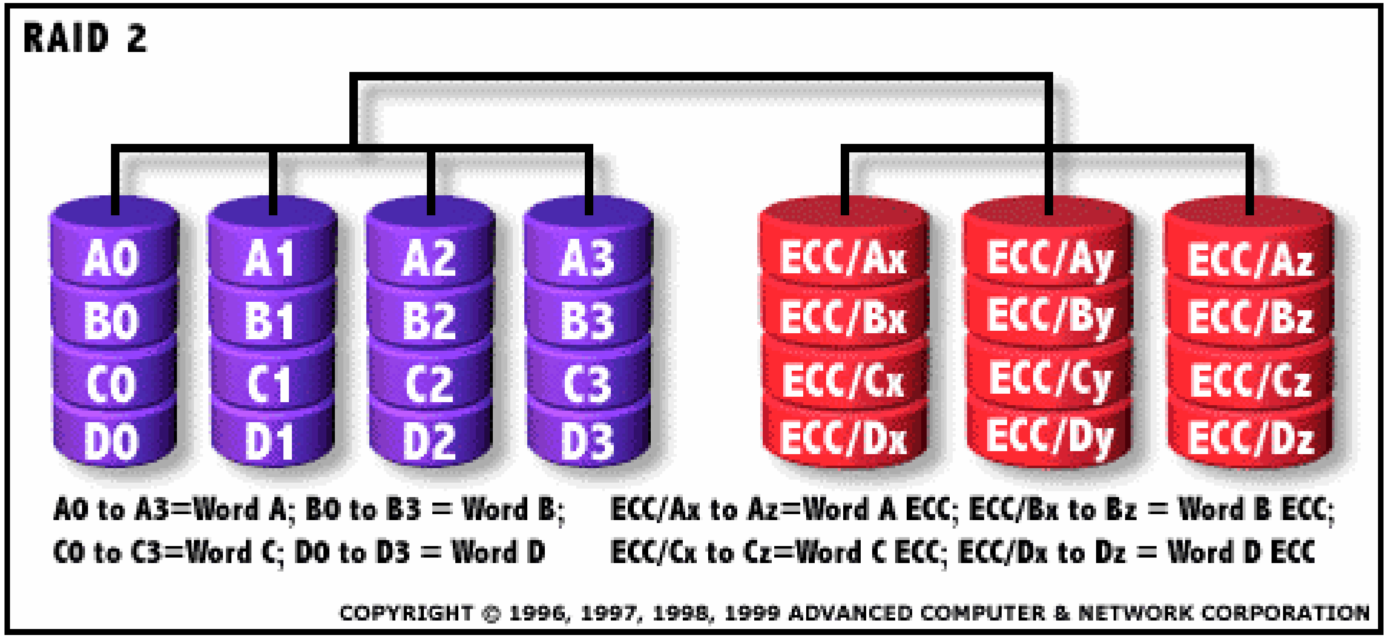


RAID 1 (mirroring) – II

- ✓ Advantages
  - one write or two reads possible per mirrored pair
  - 100% redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk
  - transfer rate per block is equal to that of a single disk
  - under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures
- ✓ Disadvantages
  - highest disk overhead of all RAID types (100%) - inefficient
  - may not support hot swap of failed disk when implemented in "software"

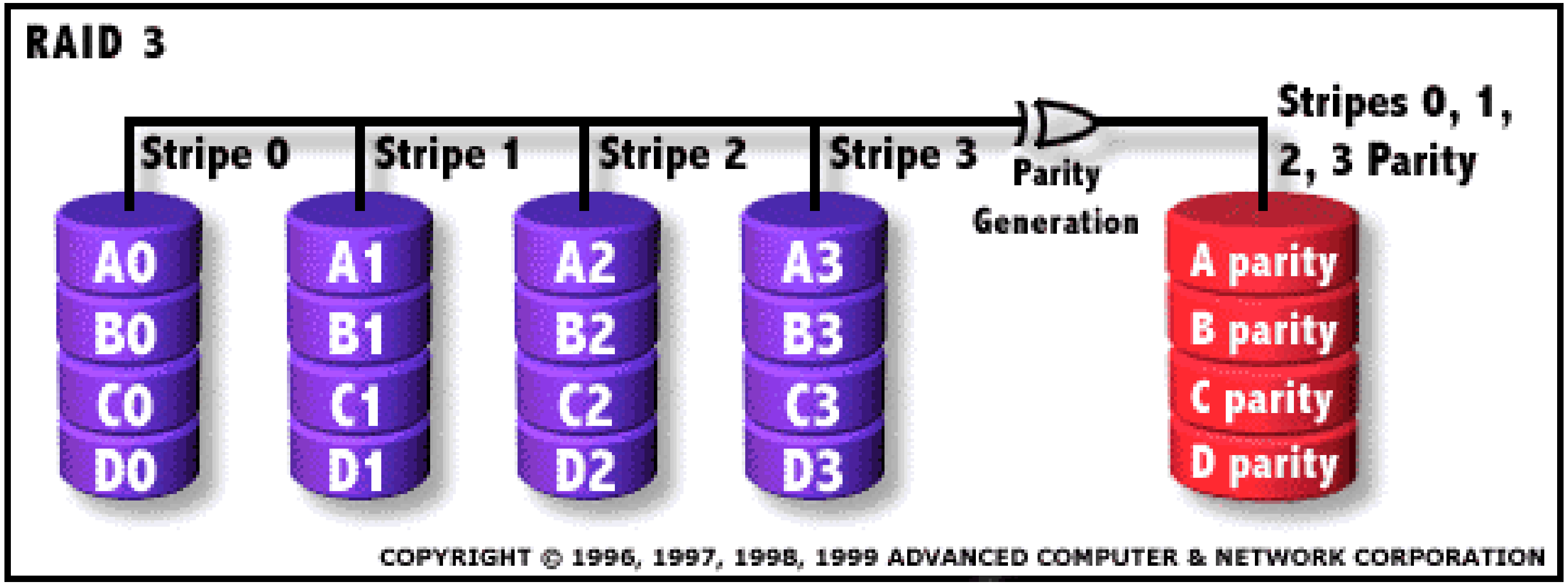
RAID 2 (Hamming ECC) – I

- ✓ RAID 2: hamming ECC
- ✓ Each bit of data word is written to a data disk drive. Each data word has its Hamming Code ECC word recorded on the ECC disks. On read, the ECC code verifies correct data or corrects single disk errors.
- ✓ NB! no commercial implementations exist



RAID 3 (Bit-Interleaved Parity) – I

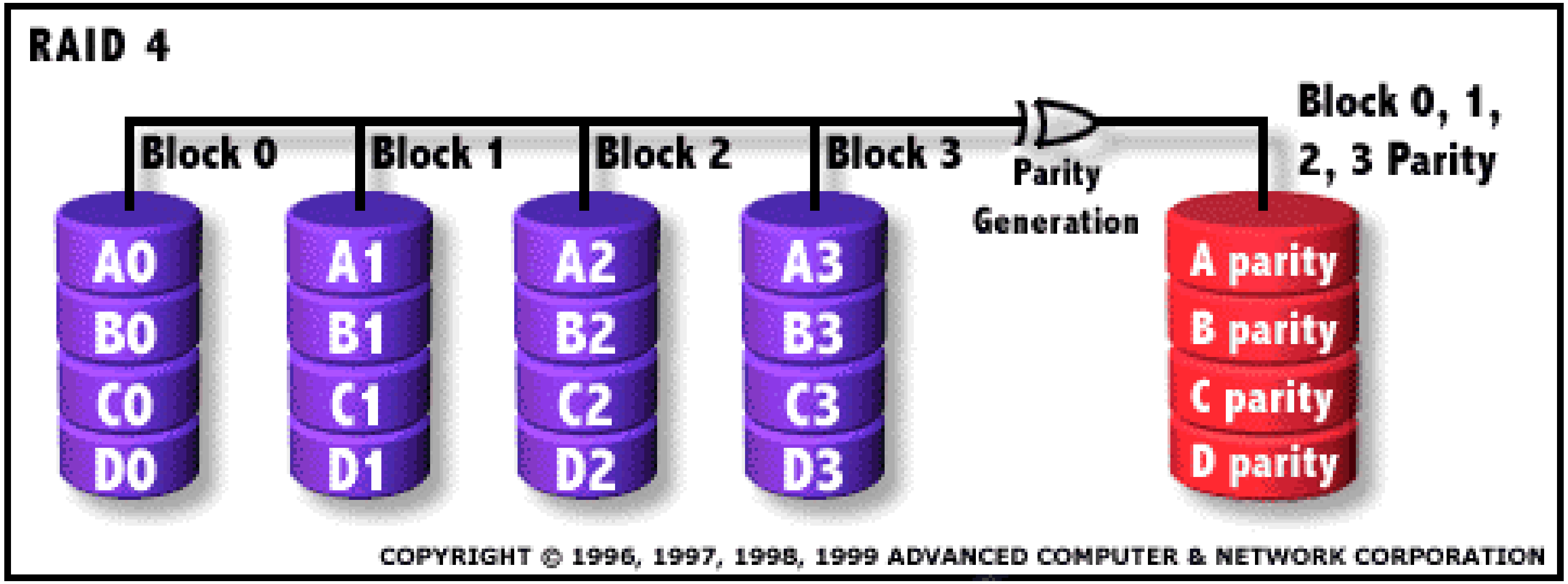
- ✓ RAID 3: parallel transfer with parity
- ✓ The data block is subdivided ("striped") and written on the data disks. Stripe parity is generated on writes, recorded on the parity disk and checked on reads.



COPYRIGHT © 1996, 1997, 1998, 1999 ADVANCED COMPUTER & NETWORK CORPORATION

RAID 4 (Block-Interleaved Parity) – I

- ✓ RAID 4: independent data disks with shared parity disk
- ✓ Each entire block is written onto one data disk. Parity for same rank blocks is generated on writes, recorded on the parity disk and checked on reads.



RAID 4 (Block-Interleaved Parity) – II

- Advantages
  - high read data transaction rate
  - low ratio of parity disks to data disks means high efficiency
- Disadvantages
  - quite complex controller design
  - difficult and inefficient data rebuild in the event of disk failure
  - updating blocks can be a bottleneck as all parity blocks are on the same disk (and must be accessed for all write operations)

RAID 4 (Block-Interleaved Parity) – III

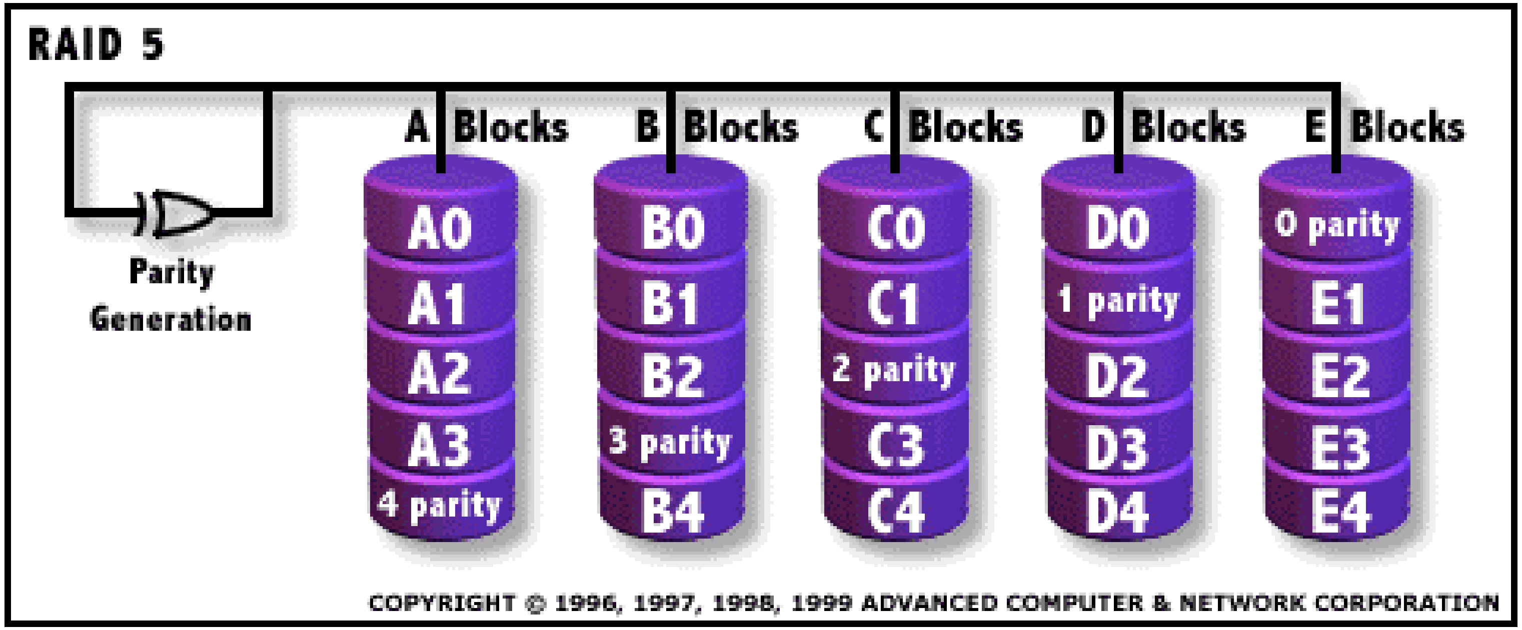
- ✓ Read operations
  - no different than normal disk reads
  - disks can be accessed in parallel
  - if requested disk is busy and all of the other disks are idle (including parity disk), we may read all other disks and generate requested block

- ✓ Write operations
  - update data block *and* parity block
  - parity block can be updated two ways
    - reading all n block and generating the whole parity block from scratch
    - perform modulo-2 sum of old and new version of the block, and simply add the sum (again using modulo-2 sum) to the parity block

- ✓ Recovery
  - Any disk can fail and be restored using modulo-2 sum of the other disks

RAID 5 (Block-Interleaved Distributed Parity) – I

- ✓ RAID 5: independent data disks with distributed parity disk
- ✓ Each entire data block is written on a data disk; parity for blocks in the same rank is generated on writes, recorded in a distributed location and checked on reads.



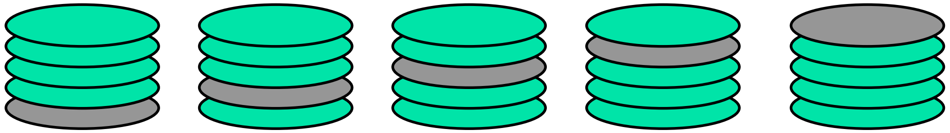
COPYRIGHT © 1996, 1997, 1998, 1999 ADVANCED COMPUTER & NETWORK CORPORATION

RAID 5 (Block-Interleaved Distributed Parity) – II

- ✓ Advantages
  - highest read data transaction rate
  - medium write data transaction rate
  - low ratio of parity disks to data disks means high efficiency
  - parity distributed, i.e., updating parity blocks goes to different disks
- ✓ Disadvantages
  - complex controller design
  - difficult to rebuild in the event of a disk failure
  - individual block data transfer rate same as single disk

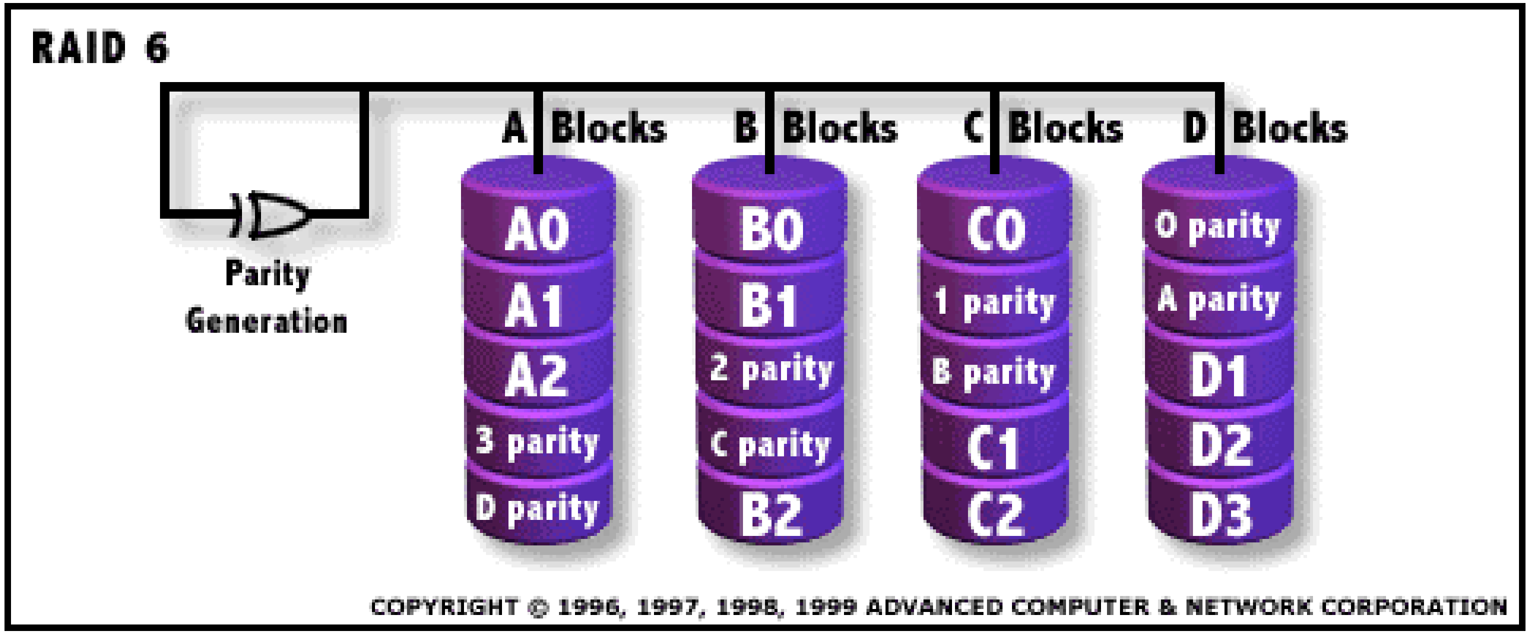
RAID 5 (Block-Interleaved Distributed Parity) – III

- ✓ Read, write, and recovery operations are analogous to RAID 4
- ✓ However, parity is distributed, possible scheme
  - ➢ n + 1 disks numbered 0 to n
  - ➢ let cylinder *i* on disk *j* be parity if the reminder of *i / (n + 1)* is *j*,
    i.e.,
    cylinder *i + 1* on disk *j + 1* be next parity



RAID 6 (P+Q Redundancy) – I

- ✓ RAID 6: independent data disks with two independent distributed parity schemes
- ✓ RAID 6 is essentially an extension of RAID level 5 which allows for additional fault tolerance by using a second independent distributed parity scheme
- ✓ Data is striped on a block level across a set of drives, just like in RAID 5, and a second set of parity is calculated and written across all the drives



COPYRIGHT © 1996, 1997, 1998, 1999 ADVANCED COMPUTER & NETWORK CORPORATION

RAID 6 (P+Q Redundancy) – II

- ✓ Advantages
  - provides for an extremely high data fault tolerance and can sustain multiple simultaneous drive failures
  - perfect solution for error-critical applications
  - parity information *can* be distributed as in RAID 5
- ✓ Disadvantages
  - very complex controller design
  - controller overhead to compute parity addresses is extremely high
  - very poor write performance
  - requires at least N+2 drives to implement because of two-dimensional parity scheme

RAID 6 (P+Q Redundancy) – III

- ✓ In general, we can add several redundancy disks to be able do deal with several simultaneous disk crashes
- ✓ Many different strategies based on different EECs, e.g.,:
  - ➢ Read-Solomon Code (or derivates):
    - corrects n simultaneous disk crashes using n parity disks
    - a bit more expensive parity calculations compared to modulo-2 sum
  - ➢ Hamming Code:
    - corrects 2 disk failures using $2^K - 1$ disks where k disks are parity disks and $2^K - k - 1$
    - uses modulo-2 sum
    - the parity disks are calculated using the data disks determined by the hamming code, i.e., a k x $(2^K - 1)$ matrix of 0's and 1's representing the $2^K - 1$ numbers written binary except 0

RAID 6 (P+Q Redundancy) – IV

- ✓ Example:
  using a Hamming code matrix, 7 disks, 3 parity disks

	disk number			
parity	7	0	0	1
	6	0	1	0
	5	1	0	0
data	4	0	1	1
	3	1	0	1
	2	1	1	0
	1	1	1	1

Note 1:
the rows represent binary numbers 1 - 7

Note 2:
the rows for the parity disks have single 1's

Note 3:
the rows for the data disks have two or more 1's

Note 4:
the idea of each column now is that the parity disk having a 1 in this column is generated using the data disks having one in this column:

- parity disk 5 is generated using disk 1, 2, 3
- parity disk 6 is generated using disk 1, 2, 4
- parity disk 7 is generated using disk 1, 3, 4

Note 5:
the parity blocks are generated using modulo-2 sum from the data blocks

RAID 6 (P+Q Redundancy) – V

- ✓ Example (cont.):

calculating parity using the hamming matrix to find the corresponding data disks to each parity disk

Hamming code matrix

parity	7	0	0	1
	6	0	1	0
	5	1	0	0
	4	0	1	1
data	3	1	0	1
	2	1	1	0
	1	1	1	1

disk block values

parity	7	
	6	
	5	
	4	01000010
data	3	00111000
	2	10101010
	1	11110000

Note 1: parity disk 5 is generated using disk 1, 2, 3
11110000 ⊕ 10101010 ⊕ 00111000 = 01100010

Note 2: parity disk 6 is generated using disk 1, 2, 4
11110000 ⊕ 10101010 ⊕ 01000010 = 00011011

Note 3: parity disk 7 is generated using disk 1, 3, 4
11110000 ⊕ 00111000 ⊕ 01000010 = 10001001

RAID 6 (P+Q Redundancy) – VI

- ✓ Read operations is performed from any data disk as a normal read operation

- ✓ Write operations are performed as shown on previous slide (similar RAID 5), but
  - ➢ now there are several parity disks
  - ➢ each parity disk does not use all data disks

- ✓ Update operations are performed as for RAID 4 or RAID 5:
  - ➢ perform modulo-2 sum of old and new version of the block, and simply add the sum (again using modulo-2 sum) to the parity block

RAID 6 (P+Q Redundancy) – VII

- ✓ Example update:
  - update data disk 2 to 00001111
  - parity disks 5 and 6 is using data disk 2

disk block values

parity	7	10001001
	6	10111110
	5	11000111
data	4	01000010
	3	00111000
	2	00001111
	1	11110000

Note 1:
old value is 10101010.
Difference is 10101010 ⊕ 00001111 = 10100101

Note 2:
insert new value in data disk 2: 00001111

Note 3:
update parity disk 5, take difference between old and new block, and perform modulo-2 sum with parity:
10100101 ⊕ 01100010 = 11000111

Note 4:
insert new value in parity disk 5: 11000111

Note 5:
parity disk 6 is similarly updated

RAID 6 (P+Q Redundancy) – VIII

- ✓ Recovery operations is performed using modulo-2 sum and the parity disks
  - one disk failure is easy – just apply one set of parity and recover
  - two disk failures a bit more tricky
    - note that all parity disk computations are different
    - we will always find one configuration where only one disk has failed
    - use this configuration to recover the failed disk
    - now there is only one failed disk, and any configuration can be used

RAID 6 (P+Q Redundancy) – IX

- ✓ Example recovery:
  - disk 2 and 5 have failed

Note 1:
there is always a column in the hamming code matrix where only one of the failed disks have a 1- value

Note 2:
column 2 use data disk 2, and no other disks have crashed, i.e., use disk 1, 4, and 6 to recover disk 2

Hamming code matrix

parity	*7*	0	0	1
	6	0	1	0
	5	1	0	0
data	*4*	0	1	1
	3	1	0	1
	2	1	1	0
	1	1	1	1

disk block values

7	10001001
6	00011011
5	???
4	01000010
3	00111000
2	???
1	11110000

Note 3:
restoring disk 2:
11110000 ⊕ 01000010 ⊕ 00011011 = 10101001

Note 4:
restoring disk 5 can now be done using column 1

Summary

- ✓ Efficient storage usage
  - ➢ block size
  - ➢ scheduling
  - ➢ adjacent blocks
  - ➢ multiple disks
  - ➢ prefetching / multiple-buffering

- ✓ Disk errors
  - ➢ new disks have few errors, but they DO occur
  - ➢ checksums

- ✓ Error recovery
  - ➢ stable storage
  - ➢ RAID