DISK FAILURE MODELS

How should systems ensure that the data written to storage is protected? What techniques are required? How can such techniques be made efficient, with both low space and time overheads? These are the topics of **Data Integrity and Protection**.

Disk fault model: Disk failure is common and one simple model is that the disk is either working or failed that can be detected easily – this the *fail-stop* model. However, in most of the cases there will be troble in accessing one or more blocks while the rest is fine – disk is partially OK (fault model partial-falure) kind of scenario. There are two common cases.

- Latent Sector Error (LSE) : It happens when a sector (or a group of sectors) is physically damaged in some way or other (R/W head accidentally touchingg the surface is common). Cosmic rays are also resposible for bit-flipping. However, the in-disk errors may be detected (and cured) by the Error Correcting Codes (ECC) in some cases. If it cannot detect then the returned block is different from what has been stored
- Block Corruption (BC) : A block (or more) may be corrupted for no discerning reason and the disk did not raise a concern when returning a block

These silent faults are realy disturbing.

More on LSE and block corruption

A detailed study has revealed the following picture on the LSE and BS. We see from the table, while buying better drives reduces the frequency of both types of problem (by about an order of magnitude), they still happen often enough that you need to think carefully about them.

	Cheap (SATA)	Costly(SCSI)
LSE	9.40%	1.40%
BC	0.5%	0.05%

Handling LSE: It is rather straightforward to handle e, as they are (by definition) easily detected. The disk simply use whatever redundancy mechanism it has to return the correct data. In a mirrored RAID, for example, the system should access the alternate copy; in a RAID-4 or RAID-5 system based on parity, the system should reconstruct the block from the other blocks in the parity group. Thus, easily detected problems such as LSEs are readily recovered through standard redundancy mechanisms.

More on LSE and block corruption...contd.

Block Corruption: Unlike latent sector errors, detection of corruption is a key problem. The primary mechanism used to preserve data integrity is called the checksum. A checksum is simply the result of a function that takes a chunk of data (say a 4KB block) as input and computes a function over said data, producing a small summary of the contents of the data (say 4 or 8 bytes). This summary is referred to as the checksum. The goal of such a computation is to enable a system to detect if data has somehow been corrupted or altered by storing the checksum with the data and then confirming upon later access that the data's current checksum matches the original storage value.

Checksum functions: (i) One common method is XOR the elements. Here is an example 0011 0110 0101 1110 1100 0100 1100 1101
1011 1010 0001 0100 1000 1010 1001 0010
1110 1100 1110 1111 0010 1100 0011 1010
0100 0000 1011 1110 1111 0110 0110 0110
Checksum after doing columnwise XOR
0010 0000 0001 1011 1001 0100 0000 0011

ii) Addition : This method is good with low computational cost. Here addition of some-subunits of the big data chunk is added together (ignoring overflow) to form the checksum. As 1-bit error is not very rare and XOR is a bit unreliable for a double bit flipping. Addition is a bit better.

More on LSE and block corruption...contd.

Fletcher Checksum: It can detect all single and double bit errors and more. It is quite simple and involves the computation of two check bytes, $s1$ and $s2$. Specifically, assume a block D consists of bytes $d_1 \ldots d_n$; $s1$ is simply defined as $s1 = s1 + d_i \ mod \ 255 (\text{computed over al } d_i)$; and $s2$ in turn is: $s2 = s2 + s1 \ mod \ 255 \ (\text{again over all } d_i)$ The fletcher checksum is known to be almost as strong as the CRC

Cyclic Redundancy Check (CRC): Name is a bit fancy but the idea is simple. All you do is treat D (data block) as if it is a large binary number (it is just a string of bits after all) and divide it by an agreed upon value (k). The remainder of this division is the value of the CRC. As it turns out, one can implement this binary modulo operation rather efficiently, and hence the popularity of the CRC in networking as well.

Whatever the method used, it should be obvious that there is no perfect checksum: it is possible two data blocks with non-identical contents will have identical checksums, something referred to as a collision.

In choosing a good checksum function, we are thus trying to find one that minimizes the chance of collisions while remaining easy to compute.

Storing the Checksum on the disk

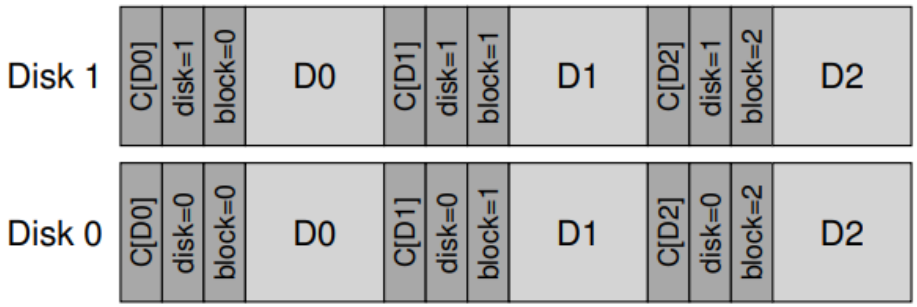
It is quite natural to compute the checksum for a sector/block and keep it separate with each sector/block. The following could be a layout (1st figure) for each 512 byte block preceded by the 2 byte checksum (C). Disks only can write in sector-sized chunks (512 bytes) or multiples thereof. One solution employed by drive manufacturers is to format the drive with 520-byte sectors to store the checksum as well. If it is 512 bytes only another layout could be (2nd figure)
In this scheme, the n checksums are stored together in a sector, followed by n data blocks, followed by another checksum sector for the next n blocks, and so forth. This scheme can be less efficient. If the FS, for example, wants to overwrite block D1, it has to read in the checksum sector containing C(D1), update C(D1) in it, and then write out the checksum sector as well as the new data block D1 (thus, one read and two writes). The earlier approach (of one checksum per sector) just performs a single write.

Misdirected writes and Lost writes

While the checksum method described earlier can mostly detect the LSE and Corruption we have other kinds of errors in modern disk systems.

Misdirected writes: This happens when the disk write the data correctly, except in the wrong location. In a single-disk system, this means that the disk wrote block Dx not to address x (as desired) but rather to address y (thus "corrupting" Dy); in addition, within a multi-disk system, the controller may also write Di,x not to address x of disk i but rather to some other disk j. The solution is simple – along with the checksum add unformation like the disk and sector number of the block.

You can see from the on-disk format that there is now a fair amount of redundancy on disk: for each block, the disk number is repeated within each block, and the offset of the block in question is also kept next to the block itself. The presence of redundant information should be no surprise, though; redundancy is the key to error detection (in this case) and recovery (in others).



Misdirected writes and Lost writes...contd

Lost writes: This happens when the disk did not write the data and returns write-complete information to the upper layer.
The obvious question here is: do any of our checksumming strategies from above (e.g., basic checksums, or physical identity) help to detect lost writes?
Unfortunately, the answer is no: the old block likely has a matching checksum, and the physical ID used above (disk number and block offset) will also be correct.
One classic approach is to perform a write verify or read-after-write; by immediately reading back the data after a write, a system can ensure that the data indeed reached the disk surface. This approach, however, is quite slow, doubling the number of I/Os needed to complete a write.
Some systems add a checksum elsewhere in the system to detect lost writes. For example, Sun's Zettabyte File System (ZFS) includes a checksum in each file system inode and indirect block for every block included within a file. Thus, even if the write to a data block itself is lost, the checksum within the inode will not match the old data.

Scrubbing and Checksum overhead

Scrubbing: Checksum operation is done when a block is in use (r/w perhaps). However, most data is rarely accessed, and thus would remain unchecked. WIth time there is a bit rot that could eventually affect all copies of a particular piece of data. To alleviate many systems utilize disk scrubbing by periodically (typically, nightly or weekly basis) reading through every block and identify data corruption; if any.

Cheksum overhead: They are two type; space and time. Space overheads come in two forms. The first is on the disk (or other storage medium) itself; each stored checksum takes up room on the disk. A typical ratio might be an 8- byte checksum per 4 KB data block, for a 0.19% on-disk space overhead. The second type of space overhead comes in the memory of the system. When accessing data, there must now be room in memory for the checksums as well as the data itself. The time overheads is noticeable. Minimally, the CPU must compute the checksum over each block, both when the data is stored as well as when it is accessed. To reduce the overhead many systems combine data copying and checksumming into one streamlined activity; because the copy is needed anyhow (e.g., to copy the data from the kernel page cache into a user buffer), combined copying/checksumming can be quite effective.