**Chapter 45: Data Integrity and Protection**

We will now investigate techniques used to ensure that the data we put into our storage system is the same when the storage system returns it to us.

**45.1 Disk Failure Modes**

The **fail-stop** model was assuming either the entire disk is working, or it fails completely, and the detection of such a failure is straightforward. Thus, this makes building RAID relatively simple.

There are other types of failure modes. Specifically, two types of single-block failures are common and worthy of consideration: **latent-sector errors (LSEs)** and **block corruption**.

Text, table

Description automatically generated

**LSEs** arise when a disk sector (or group of sectors) has been damaged in some way. For example, if the disk head touches the surface for some reason, it may damage the surface, making the bits unreadable. **Cosmic rays** can also flip bits, leading to incorrect contents. Fortunately, in-disk error correcting codes (ECC) are used by the drive to determine whether the on-disk bits in a block are good, and in some cases, to fix them; if they are not good, and the drive does not have enough information to fix the error, the disk will return an error when a request is issued to read them.

There can be cases where a disk block becomes **corrupt** in a way not detectable by the disk itself. These types of faults are particularly insidious because they are **silent faults**; the disk gives no indication of the problem when returning the faulty data.

**Fail-partial** disk failure model is the mode where disks can still fail in their entirely. However, disks can also seemingly be working and have one or more blocks become inaccessible (i.e., LSEs) or hold the wrong contents (i.e., corruption). Thus, when accessing a seemingly-working disk, it might return an error or wrong data once in a while.

Some findings about LSEs:

Text

Description automatically generated

Some findings about corruption:

Text

Description automatically generated

**45.2 Handling Latent Sector Errors**

When a storage system tries to access a block, and the disk returns an error, the storage system should simply use whatever redundancy mechanism it has to return the correct data. For example, in a mirrored RAID, it should access the copy, while in RAID4 or 5, we can reconstruct the block.

When an entire disk fails, the RAID tries to reconstruct the disk by reading through all the other disks in the parity group and recomputing the missing values. If, during **reconstruction**, an LSE is encountered on any one of the other disks, we have a problem: the reconstruction cannot successfully complete.

To combat this issue, some systems add an extra degree of redundancy. For example, NetApp’s RAID-DP has the equivalent of two parity disks instead of one. When an LSE is discovered during reconstruction, the extra parity helps to reconstruct the missing block. As always, there is a cost, in that maintaining two parity blocks for each stripe is more costly.

**45.3 Detecting Corruption: The Checksum**

We will handle silent failures via data corruption. Unlike latent sector errors, detection of corruption is a key problem.

The primary mechanism used by modern storage systems 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.

**Common Checksum Functions**

A **trade-off** that is common in systems arises here: usually, the more protection you get, the costlier it is.

One simple checksum function that some use is based on exclusive or (XOR). With XOR-based checksums, the checksum is computed by XOR’ing each chunk of the data block being checksummed, thus producing a single value that represents the XOR of the entire block.

For example, the 16 data bytes:



In binary, it is:

Text

Description automatically generated with medium confidence

If we use XOR by column, we will get:



Which is 0x201b9403. There is limitation as if two bits in the same position is changed, we cannot detect the corruption.

Another basic checksum function is **addition**. This approach has the advantage of being fast; computing it just requires performing 2’s-complement addition over each chunk of the data, ignoring overflow. It can detect many changes in data, but is not good if the data, for example, is shifted.

A slightly more complex algorithm is known as the **Fletcher checksum**. It is quite simple to compute and involves the computation of two check bytes, s1 and s2. Specifically, assume a block D consists of bytes d1 ... dn; s1 is defined as follows: s1 = (s1 + di) mod 255 (computed over all di); s2 in turn is: s2 = (s2 + s1) mod 255 (again over all di). This method can detect all single-bit, double-bit errors and many burst errors.

A stronger checksum is **cyclic redundancy check (CRC)**. Assume you wish to compute the checksum over a data block D. All you do is treat D 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.

However, 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**.

**Checksum Layout**

The most basic approach simply stores a checksum with each disk sector (or block). Given a data block D, let us call the checksum over that data C(D). Thus, without checksums, the disk layout looks like this:

Table

Description automatically generated

With checksums, the layout adds a single checksum for every block:

Diagram

Description automatically generated with medium confidence

However, it is difficult to achieve such layout. Instead, we can have an entire block for checksums:

A picture containing diagram

Description automatically generated

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 approach fits the working on all disks, but less efficient. If the file system, 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 and new data block D1 (thus, one read and two writes). The earlier approach (of one checksum per sector) just performs a single write.

**45.4 Using Checksums**

When reading a block D, the client (i.e., file system or storage controller) also reads its checksum from disk Cs(D), which we call the **stored checksum** (hence the subscript Cs). The client then computes the checksum over the retrieved block D, which we call the **computed checksum** Cc(D). At this point, the client compares the stored and computed checksums. If they are different, the data has changed, and we have a corruption.

Given such corruption, we can either return its copy (if exists) or return error.

**45.5 A New Problem: Misdirected Writes**

The first failure mode of interest is called a **misdirected write**. This arises in disk and RAID controllers which write the data to disk 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.

To solve this, we add a little more information to each checksum, which is **physical identifier (physical ID)**

A screenshot of a computer

Description automatically generated with low confidence

If the client is reading block 4 on disk 10 (D10,4), the stored information should include that disk number and sector offset, as shown above. If the information does not match, a misdirected write has taken place, and a corruption is now detected.

The presence of redundant information should be no surprise, though; redundancy is the key to error detection (in this case) and recovery (in others). A little extra information, while not strictly needed with perfect disks, can go a long ways in helping detect problematic situations should they arise.

**45.6 One Last Problem: Lost Writes**

**Lost write** occurs when the device informs the upper layer that a write has completed but in fact it never is persisted; thus, what remains is the old contents of the block rather than the updated new contents.

Checksum now cannot help since the old block likely has a matching checksum and the physical ID used above is also correct.

One classic approach is to perform a **write verify** or **read-after-write** by immediately check the data after a write. This approach is costly since we have to double the number of I/Os needed for 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. Only if the writes to both the inode and the data are lost simultaneously will such a scheme fail, an unlikely (but unfortunately, possible!) situation.

**45.7 Scrubbing**

When will checksums be checked?

Unchecked data is problematic for a reliable storage system, as bit rot could eventually affect all copies of a particular piece of data. To remedy this problem, many systems utilize disk scrubbing of various forms. By periodically reading through every block of the system, and checking whether checksums are still valid, the disk system can reduce the chances that all copies of a certain data item become corrupted. Typical systems schedule scans on a nightly or weekly basis.

**45.8 Overheads Of Checksumming**

Space overheads come in two forms:

1. The first is on the disk (or other storage medium) itself; each stored checksum takes up room on the disk, which can no longer be used for user data. A typical ratio might be an 8- byte checksum per 4 KB data block, for a 0.19% on-disk space overhead.
2. 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. However, if the system simply checks the checksum and then discards it once done, this overhead is short-lived and not much of a concern. If checksums are kept in memory, there will be a small overhead.

The time overheads induced by checksumming can be quite noticeable. Minimally, the CPU must compute the checksum over each block, both when the data is stored and when it is accessed. One approach to reducing CPU overheads, employed by many systems that use checksums (including network stacks), is to combine data copying and checksumming into one streamlined activity.

Some checksumming schemes can induce extra I/O overheads, particularly when checksums are stored distinctly from the data (thus requiring extra I/Os to access them), and for any extra I/O needed for background scrubbing.