

Data Integrity and Protection

Beyond the basic advances found in the file systems we have studied thus far, a number of features are worth studying. In this chapter, we focus on reliability once again (having previously studied storage system reliability in the RAID chapter). Specifically, how should a file system or storage system ensure that data is safe, given the unreliable nature of modern storage devices?

This general area is referred to as **data integrity** or **data protection**. Thus, we will now investigate techniques used to ensure that the data you put into your storage system is the same when the storage system returns it to you.

CRUX: HOW TO ENSURE DATA INTEGRITY

How should systems ensure that the data written to storage is protected? What techniques are required? How can such techniques be made efficient?

43.1 Disk Failure Modes

As you learned in the chapter about RAID, disks are not perfect, and can fail (on occasion). In early RAID systems, the model of failure was quite simple: either the entire disk is working, or it fails completely, and the detection of such a failure is straightforward. This **fail-stop** model of disk failure makes building RAID relatively simple [S90].

What you didn't learn is about all the other types of failure modes modern disks exhibit. Specifically, as Bairavasundaram et al. studied in great detail [B+07, B+08], modern disks will occasionally seem to be mostly working but have trouble correctly accessing one or more blocks. Specifically, two types of single-block failures are common: **latent-sector errors (LSEs)** and **block corruption**. We'll now discuss each in more detail.

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 (a **head crash**, something which shouldn't happen during normal operation), 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; if they are not, the disk will return an error when a request is issued to read them.

There are also cases where a disk block becomes **corrupt** in a way not detectable by the disk itself. For example, buggy disk firmware may write a block to the wrong location; in such a case, the disk ECC indicates the block contents are fine, but from the client's perspective the wrong block is returned when subsequently accessed. Similarly, a block may get corrupted when it is transferred from the host to the disk across a faulty bus; the resulting corrupt data is stored by the disk, but it is not what the client desires. 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.

Prabhakaran et al. describes this more modern view of disk failure as the **fail-partial** disk failure model [P+05]. In this view, disks can still fail in their entirety (as was the case in the traditional fail-stop model); 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, once in a while it may either return an error when trying to read or write a given block (a non-silent partial fault), and once in a while it may simply return the wrong data (a silent partial fault).

Both of these types of faults are somewhat rare, but just how rare? Table 43.1 summarizes some of the findings from the two Bairavasundaram studies [B+07,B+08].

The table shows the percent of drives that exhibited at least one LSE or block corruption over the course of the study (about 3 years,

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

Table 43.1: Frequency of LSEs and Block Corruption

over 1.5 million disk drives). The table further sub-divides the results into “cheap” drives (usually SATA drives) and “costly” drives (usually SCSI or FibreChannel). As you can 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.

Some additional findings about LSEs are:

- Costly drives with more than one LSE are as likely to develop additional errors as cheaper drives
- For most drives, annual error rate increases in year two
- LSEs increase with disk size
- Most disks with LSEs have less than 50
- Disks with LSEs are more likely to develop additional LSEs
- There exists a significant amount of both spatial and temporal locality
- Disk scrubbing is useful (most LSEs were found this way)

Some findings about corruption:

- Chance of corruption varies greatly across models within same drive class
- Age affects are different across models
- Workload and disk size have little impact on corruption
- Most disks with corruption only have a few corruptions
- Corruption is not independent with a disk, or across disks in a RAID
- There exists spatial locality, and some temporal locality
- There is a weak correlation with LSEs

To learn more about these failures, you should likely read the original papers [B+07,B+08]. But hopefully the main point should be clear: if you really wish to build a reliable storage system, you must include machinery to detect and recovery from both LSEs and block corruption.

43.2 Handling Latent Sector Errors

Given these two new modes of partial disk failure, we should now try to see what we can do about them. Let's first tackle the easier of the two, namely latent sector errors.

CRUX: HOW TO HANDLE LATENT SECTOR ERRORS

How should a storage system handle the presence of latent sector errors? How much extra machinery is needed to handle this form of partial failure?

As it turns out, latent sector errors are rather straightforward to handle, as they are (by definition) easily detected. 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. 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.

The growing prevalence of LSEs has influenced RAID designs over the years. One particularly interesting problem arises in RAID-4/5 systems when both full-disk faults and LSEs occur in tandem. Specifically, when an entire disk fails, the RAID tries to **reconstruct** the disk (say, onto a hot spare) by reading through all of 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 [C+04]. 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; however, the log-structured nature of the NetApp **WAFL** file system mitigates that cost in many cases [HLM94]. The remaining cost is space, in the form of an extra disk for the second parity block.

43.3 Detecting Corruption: The Checksum

Let's now tackle the more challenging problem, that of silent failures via data corruption. How can we prevent users from getting bad data when corruption arises, and thus leads to disks returning bad data?

CRUX: HOW TO PRESERVE DATA INTEGRITY DESPITE CORRUPTION

Given the silent nature of such failures, what can a storage system do to detect when corruption arises? What techniques are needed? How can one implement them efficiently?

Unlike latent sector errors, *detection* of corruption is a key problem. How can a client tell that a block has gone bad? Once it is known that a particular block is bad, *recovery* is the same as before: you need to have some other copy of the block around (and hopefully, one that is not corrupt!). Thus, we focus here on detection techniques.

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 number of different functions are used to compute checksums, and vary in strength (i.e., how good they are at protecting data integrity) and speed (i.e., how quickly can they be computed). A trade-off that is common in systems arises here: usually, the more protection you get, the costlier it is. There is no such thing as a free lunch.

TIP: THERE’S NO FREE LUNCH

There’s No Such Thing As A Free Lunch, or TNSTAAFL for short, is an old American idiom that implies that when you are seemingly getting something for free, in actuality you are likely paying some cost for it. It comes from the old days when diners would advertise a free lunch for customers, hoping to draw them in; only when you went in, did you realize that to acquire the “free” lunch, you had to purchase one or more alcoholic beverages. Of course, this may not actually be a problem, particularly if you are an aspiring alcoholic (or typical undergraduate student).

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

To make this more concrete, imagine we are computing a 4-byte checksum over a block of 16 bytes (this block is of course too small to really be a disk sector or block, but it will serve for the example). The 16 data bytes, in hex, look like this:

365e c4cd ba14 8a92 ecef 2c3a 40be f666

If we view them in binary, we get the following:

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

Because we’ve lined up the data in groups of 4 bytes per row, it is easy to see what the resulting checksum will be: simply perform an XOR over each column to get the final checksum value:

0010 0000 0001 1011 1001 0100 0000 0011

The result, in hex, is 0x201b9403.

XOR is a reasonable checksum but has its limitations. If, for example, two bits in the same position within each checksummed unit change, the checksum will not detect the corruption. For this reason, people have investigated other checksum functions.

Another simple checksum function is addition. This approach has the advantage of also 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**, named (as you might guess) for the inventor, John G. Fletcher [F82]. It is quite simple and involves the computation of two check bytes, s_1 and s_2 . Specifically, assume a block D consists of bytes $d_1 \dots d_n$; s_1 is simply defined as follows: $s_1 = s_1 + d_i \bmod 255$ (computed over all d_i); s_2 in turn is: $s_2 = s_2 + s_1 \bmod 255$ (again over all d_i) [F04]. The fletcher checksum is known to be almost as strong as the CRC (described next), detecting all single-bit errors, all double-bit errors, and a large percentage of burst errors [F04].

One final commonly-used checksum is known as a **cyclic redundancy check (CRC)**. While this sounds fancy, the basic idea is quite simple. 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. As it turns out, one can implement this binary modulo operation rather efficiently, and hence the popularity of the CRC in networking as well. See elsewhere for more details [M13].

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**. This fact should be intuitive: after all, computing a checksum is taking something large (e.g., 4KB) and producing a summary that is much smaller (e.g., 4 or 8 bytes). In choosing a good checksum function, we are thus trying to find one that minimizes the chance of collisions while remaining easy to compute.

Checksum Layout

Now that you understand a bit about how to compute a checksum, let's next analyze how to use checksums in a storage system. The first question we must address is the layout of the checksum, i.e., how should checksums be stored on disk?

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:

| D0 | D1 | ...

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

| C(D0) | D0 | C(D1) | D1 | ...

Because checksums are usually small (e.g., 8 bytes), and disks only can write in sector-sized chunks (512 bytes) or multiples thereof, one problem that arises is how to achieve the above layout. One solution employed by drive manufacturers is to format the drive with 520-byte sectors; an extra 8 bytes per sector can be used to store the checksum.

In disks that don't have such functionality, the file system must figure out a way to store the checksums packed into 512-byte blocks. One such possibility is as follows:

| C(D0) | C(D1) | ... | C(Dn-1) | D0 | D1 | ...

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 has the benefit of working on all disks, but can be 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 as well as the new data block $D1$ (thus, one read and two writes). The earlier approach which adds the checksum to every sector just has to perform a single write.

43.4 Using Checksums

With a checksum layout decided upon, we can now proceed to actually understand how to *use* the checksums. It is, as you might have guessed, straightforward.

When reading a block D , the client (i.e., file system or storage controller) also reads in its checksum from disk $C_s(D)$, which we refer to as the **stored checksum** (hence the subscript C_s). The client then computes the checksum over the stored data block D , which we call the **computed checksum** $C_c(D)$. At this point, the client compares

the stored and computed checksums; if they are equal (i.e., $C_s(D) == C_c(D)$), the data has likely not been corrupted, and thus can be safely returned to the user. If they do *not* match (i.e., $C_s(D) != C_c(D)$), this implies the data has somehow changed since the time it was stored (since the stored checksum reflects the value of the data at that time). In this case, we have a corruption, which our checksum scheme has helped us to detect.

Given a corruption, the natural question is what should we do about it? If the storage system has a redundant copy, the answer is easy: try to use it instead. If the storage system has no such copy, the likely answer is to return an error. In either case, realize that corruption detection is not a magic bullet; if there is no other way to get the non-corrupted data, you are simply out of luck.

43.5 A New Problem: Misdirected Writes

The basic scheme described above works well in the general case of corrupted blocks. However, modern disks have a couple of unusual failure modes that require different solutions.

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 D_x not to address x (as desired) but rather to address y (thus “corrupting” D_y); in addition, within a multi-disk system, the controller may also write $D_{i,x}$ not to address x of disk i but rather to some other disk j . Thus our question:

CRUX: HOW TO HANDLE MISDIRECTED WRITES

How should a storage system or disk controller detect misdirected writes? What additional features are required from the checksum?

The answer is not surprisingly simple: add a little more information to each checksum. In this case, adding a **physical identifier** (**physical ID**) is quite helpful. For example, if the stored information now contains the checksum $C(D)$ as well as the disk and sector number of the block, it is easy for the client to determine whether the correct information resides within the block. Specifically, if the client is reading block 4 on disk 10 ($D_{10,4}$), the stored information should

include that disk number and sector offset, as shown below. If the information does not match, a misdirected write has taken place, and a corruption is now detected. Here is an example of what this added information would look like on a two-disk system:

C (D0) , d=0, b=0	D0	C (D1) , d=0, b=1	D1	...
C (D0) , d=1, b=0	D0	C (D1) , d=1, b=1	D1	...

You can see from the on-disk format that there is now a lot of redundancy on disk, in that for each block the disk number and offset are stored. That should be no surprise, though; redundancy is the key to detection. A little extra information, while not strictly needed with perfect disks, can go a long ways in helping detect problematic situations should they arise.

43.6 One Last Problem: Lost Writes

Unfortunately, misdirected writes are not the last problem we will address. Specifically, some modern storage devices also have an issue known as a **lost write**, which occurs when the device informs the upper layer that a write has completed but in fact it never is persisted; thus, what remains is left is the old contents of the block rather than the updated new contents.

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. Thus our final problem:

CRUX: HOW TO HANDLE LOST WRITES

How should a storage system or disk controller detect lost writes? What additional features are required from the checksum?

There are a number of possible solutions that can help [K+08]. One classic approach [BS04] 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. Only if the writes to both the inode and the data are lost simultaneously will such a scheme fail.

43.7 Scrubbing

Given all of this discussion, you might be wondering: when do these checksums actually get checked? Of course, some amount of checking occurs when data is accessed by applications, but most data is rarely accessed, and thus would remain unchecked. 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 [K+08]. 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.

43.8 Summary

We have discussed data protection in modern storage systems, focusing on checksum implementation and usage. Different checksums protect against different types of faults; as storage devices evolve, new failure modes will arise, perhaps forcing the research community and industry to revisit some of these basic approaches or invent entirely new approaches altogether.

References

- [B+08] "An Analysis of Data Corruption in the Storage Stack"
Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '08, San Jose, CA, February 2008
The first paper to truly study disk corruption in great detail, focusing on how often such corruption occurs over three years for over 1.5 million drives. Lakshmi did this work while a graduate student at Wisconsin under our supervision, but also in collaboration with his colleagues at NetApp where he was an intern for multiple summers. A great example of how working with industry can make for much more interesting and relevant research.
- [BS04] "Commercial Fault Tolerance: A Tale of Two Systems"
Wendy Bartlett, Lisa Spainhower
IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January 2004
This classic in building fault tolerant systems is an excellent overview of the state of the art from both IBM and Tandem. Another must read for those interested in the area.
- [C+04] "Row-Diagonal Parity for Double Disk Failure Correction"
Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, Sunitha Sankar
FAST '04, San Jose, CA, February 2004
An early paper on how extra redundancy helps to solve the combined full-disk-failure/partial-disk-failure problem. Also a nice example of how to mix more theoretical work with practical.
- [F04] "Checksums and Error Control"
Peter M. Fenwick
Available: www.cs.auckland.ac.nz/compsci314s2c/resources/Checksums.pdf
A great simple tutorial on checksums, available to you for the amazing cost of free.
- [F82] "An Arithmetic Checksum for Serial Transmissions"
John G. Fletcher
IEEE Transactions on Communication, Vol. 30, No. 1, January 1982
Fletcher's original work on his eponymous checksum. Of course, he didn't call it the Fletcher checksum, rather he just didn't call it anything, and thus it became natural to name it after the inventor. So don't blame old Fletch for this seeming act of braggadocio.
- [HLM94] "File System Design for an NFS File Server Appliance"
Dave Hitz, James Lau, Michael Malcolm
USENIX Spring '94
The pioneering paper that describes the ideas and product at the heart of NetApp's core. Based on this system, NetApp has grown into a multi-billion dollar storage company. If you're interested in learning more about its founding, read Hitz's autobiography "How to Castrate a Bull: Unexpected Lessons on Risk, Growth, and Success in Business" (which is the actual title, no joking). And you thought computers had little to do with bull castration!

[K+08] “Parity Lost and Parity Regained”

Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '08, San Jose, CA, February 2008

This work of ours, joint with colleagues at NetApp, explores how different checksum schemes work (or don't work) in protecting data. We reveal a number of interesting flaws in current protection strategies, some of which have led to fixes in commercial products.

[M13] “Cyclic Redundancy Checks”

Author Unknown

Available: <http://www.mathpages.com/home/kmath458.htm>

Not sure who wrote this, but a super clear and concise description of CRCs is available here. The internet is full of information, as it turns out.

[P+05] “IRON File Systems”

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '05, Brighton, England, October 2005

Our paper on how disks have partial failure modes, which includes a detailed study of how file systems such as Linux ext3 and Windows NTFS react to such failures. As it turns out, rather poorly! We found numerous bugs, design flaws, and other oddities in this work. Some of this has fed back into the Linux community, thus helping to yield a new more robust group of file systems to store your data.

[RO91] “Design and Implementation of the Log-structured File System”

Mendel Rosenblum and John Ousterhout

SOSP '91, Pacific Grove, CA, October 1991

Another reference to this ground-breaking paper on how to improve write performance in file systems.

[S90] “Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial”

Fred B. Schneider

ACM Surveys, Vol. 22, No. 4, December 1990

This classic paper talks generally about how to build fault tolerant services, and includes many basic definitions of terms. A must read for those building distributed systems.