

The basics of file systems



Backblaze report on HDD failures in 2023

Among the most often used disks there has been the following number of failures:

Model	# of disks	% of failed disks
HGST HMS5C4040BLE640	10260	0.38%
HGST HUH721212ALE604	13144	0.95%
HGST HUH721212ALN604	10532	3.69%
WDC WUH721816ALE6L4	21607	0.30%
Toshiba MG07ACA14TA	37913	1.12%
Toshiba MGO8ACA16TA	33750	1.09%
Seagate ST4000DM000	11391	3.32%
Seagate ST8000NM0055	13914	3.85%
Seagate ST12000NM0008	19449	2.93%
Seagate ST12000NM001G	13108	1.19%
Seagate ST14000NM001G	10693	1.80%
Seagate ST16000NMO01G	27433	0.70%

A CERN experiment to measure the reliability of disks

1. An application writes 1Gb to a disk this way:
 - Choose 1k random locations on the disk, each 1Mb long,
 - Write 1Mb to the first location,
 - Wait 1s,
 - Repeat to fill remaining locations.
2. The application was run on a cluster that had approx. 3000 hosts. Each host had a HW RAID.
3. Wait 3 weeks.
4. Read blocks that were written at step 1.

A CERN experiment to measure the reliability of disks

1. An application writes 1Gb to a disk this way:
 - Choose 1k random locations on the disk, each 1Mb long,
 - Write 1Mb to the first location,
 - Wait 1s,
 - Repeat to fill remaining locations.
2. The application was run on a cluster that had approx. 3000 hosts. Each host had a HW RAID.
3. Wait 3 weeks.
4. Read blocks that were written at step 1.

CERN have found approx. 0.005% blocks that would read back data that differed from data originally written to them. Yet, all reads were reported as successful both by the hardware and by the OS.

A CERN experiment to measure the reliability of disks

Conclusions:

- Data may not be stored without redundancy.
- Applications and file systems must implement checksumming and consistency checks.
- Storage systems need to run proactive background consistency checks.
- Data replication or Reed-Solomon coding.
- ZFS and btrfs store cryptographic hashes of all blocks (ext4 has only CRC).
- Online scrubbing & repair в ZFS and XFS and in HW RAID controllers.

A CERN experiment to measure the reliability of disks

Conclusions:

- Data may not be stored without redundancy.
- Applications and file systems must implement checksumming and consistency checks.
- Storage systems need to run proactive background consistency checks.
- Data replication or Reed-Solomon coding.
- ZFS and btrfs store cryptographic hashes of all blocks (ext4 has only CRC).
- Online scrubbing & repair в ZFS and XFS and in HW RAID controllers.

Note: hard drives are not the only component prone to data corruption. PCI-e relies heavily on error correction:

<https://pcisig.com/pcie%C2%AE-60-specification-webinar-qa-error-detection-and-correction-fec>.

The same is true for ethernet, for RAM, etc.

Wrong ways to handle FS corruption

The paper [1] ran an experiment where the data of the following applications was located on a file system that would randomly corrupt the content of blocks:

- Redis,
- ZooKeeper,
- Cassandra,
- Kafka,
- RethinkDB,
- LogCabin.

[1] <https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf>

[2] <https://www.usenix.org/system/files/conference/fast18/fast18-alagappan.pdf>

Wrong ways to handle FS corruption

The paper [1] ran an experiment where the data of the following applications was located on a file system that would randomly corrupt the content of blocks:

- Redis,
 - Does not validate checksums of user data,
 - Replicates corrupted data across nodes of a cluster,
 - Corruptions of a FS are “handled” with assert()s.
- ZooKeeper,
- Cassandra,
- Kafka,
- RethinkDB,
- LogCabin.

[1] <https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf>

[2] <https://www.usenix.org/system/files/conference/fast18/fast18-alagappan.pdf>

Wrong ways to handle FS corruption

The paper [1] ran an experiment where the data of the following applications was located on a file system that would randomly corrupt the content of blocks:

- Redis,
- ZooKeeper,
 - Validates checksums in all data, but does it inside assert()s,
 - Uses Adler32 as the checksum,
- Cassandra,
- Kafka,
- RethinkDB,
- LogCabin.

[1] <https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf>

[2] <https://www.usenix.org/system/files/conference/fast18/fast18-alagappan.pdf>

Wrong ways to handle FS corruption

The paper [1] ran an experiment where the data of the following applications was located on a file system that would randomly corrupt the content of blocks:

- Redis,
- ZooKeeper,
- Cassandra,
 - Has no checksums for uncompressed data,
 - When the checksum does not match, Cassandra chooses the last write as “the correct one”. This way it can replicate corrupted data to other cluster nodes.
- Kafka,
- RethinkDB,
- LogCabin.

[1] <https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf>

[2] <https://www.usenix.org/system/files/conference/fast18/fast18-alagappan.pdf>

Ways to check the data integrity

So far, we've mentioned two mechanism of integrity checks:

- Cyclic redundancy checks,
- Cryptographic hash sums.

Let us consider them in more detail.

Cyclic Redundancy Check

Let us regard a message as a sequence of bits (elements of $GF(2)$). There is a bijection between messages and polynomials in $GF(2)[X]$:

$$a_{n-1}a_{n-2} \dots a_0 \leftrightarrow M(X) = X^n + a_{n-1}X^{n-1} + a_{n-2}X^{n-2} + \dots + a_0$$

Take a polynomial $C \in GF(2)[X]$ with $\deg C = d$, and let $r(X) = M(X) * X^d \bmod C(X)$.

$r(X)$ is called the CRC of M .

Now consider the polynomial

$$M(X) * X^d + r(X)$$

It encodes a message that is obtained from $a_{n-1}a_{n-2} \dots a_0$ by appending d bits that are the coefficients of r .

The CRC are very well suited to be implemented in the hardware. One only needs XORs and shifts to calculate a CRC.

A polynomial $C(X)$ is picked to detect specific kinds of bit errors.

Cyclic Redundancy Check, exercises

- If $C(X)$ has ≥ 2 non-zero coefficients then it detects any 1-bit flip.
- If $C(X)$ has an irreducible factor of degree m then it detects any error that changes 2 bits that are less than m places apart.
- If $C(X)$ is divisible by $X + 1$ then it detects any error that changes an odd number of bits.

Cyclic Redundancy Check

The CRC is often used this way:

```
struct something
{
    some fields
    ...
    u32 crc;
}
```

1. Calculate the CRC of all struct fields except `->crc`,
2. Set `->crc` to a value such that the CRC of the whole struct be 0.

Exercise: let us have a message M and a generator polynomial $C(X)$ that has the degree d . Find a d -bits integer X such that $CRC(concat(M, X)) = 0$.

Examples of CRC

Name	Applications	The generator polynomial*
CRC-16-CCITT	Bluetooth	0x1021
CRC-16-IBM	USB	0x8005
CRC-32	Ethernet, SATA, MPEG-2, gzip, bzip2, PNG	0x04C11DB7
CRC-32C (Castagnoli)**	iSCSI, SCTP, ext4, btrfs, Ceph	0x1EDC6F41

* Bits in these numbers are used as the coefficients of the generator polynomial.

** SSE4.2 has dedicated instructions to calculate this CRC.

Cryptographic hashes

CRCs are very fast to calculate and detect some most frequent corruptions. However, they are easily fooled by deliberately introduced errors.

- Redis,
- ZooKeeper,
 - Validates checksums in all data, but does it inside assert()s,
 - Uses **Adler32** as the checksum,
- Cassandra,
- Kafka,
- RethinkDB,
- LogCabin.

Adler32 was designed to detect typical corruptions in data compressors. It is good only for short bit strings. It cannot reliably detect corruptions in longer blocks. That is why ZooKeeper may believe corrupted blocks to be valid.

Cryptographic hashes

CRCs are very fast to calculate and detect some most frequent corruptions. However, they are easily fooled by deliberately introduced errors.

To verify reliably that data was not tampered with or corrupted, we use cryptographic hashes:

- MD5,
- SHA1,
- SHA-256, SHA-384, SHA-512.

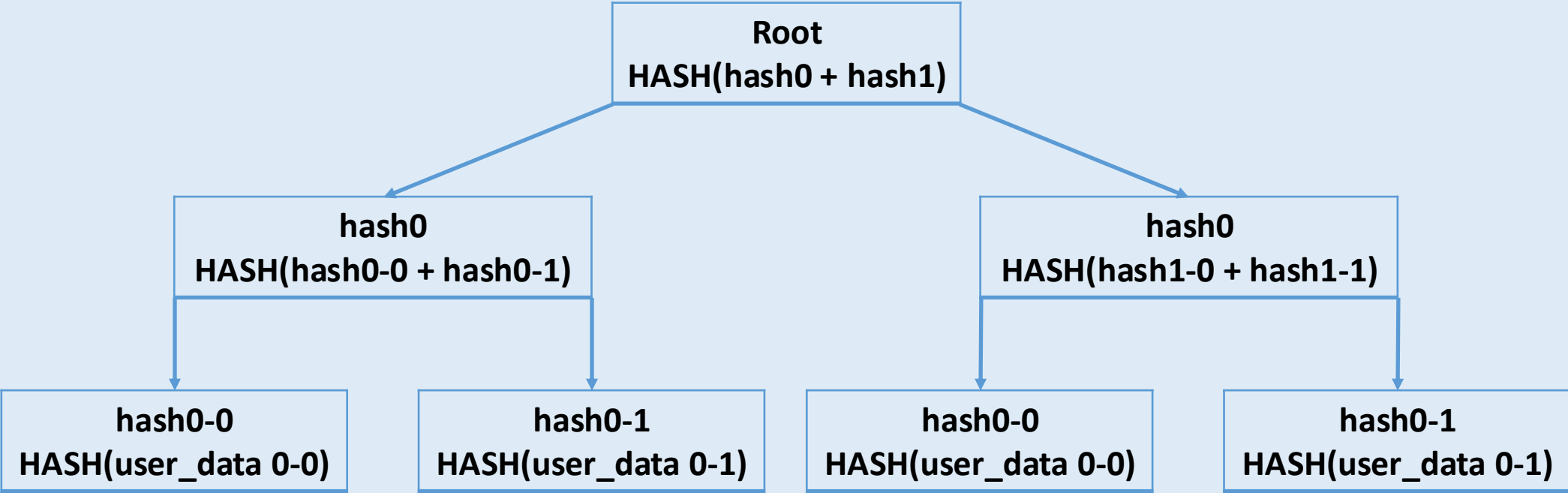
These algorithms may be relied upon because we know no algorithm for finding hash collisions except brute-force search. However,

- The search for collisions in MD5 is no longer “too computationally expensive”,
- For SHA1, a collision was found and there is a way to construct more collisions using that one,

SHA-256 and more modern hashes still need a full brute-force search to find a collision.

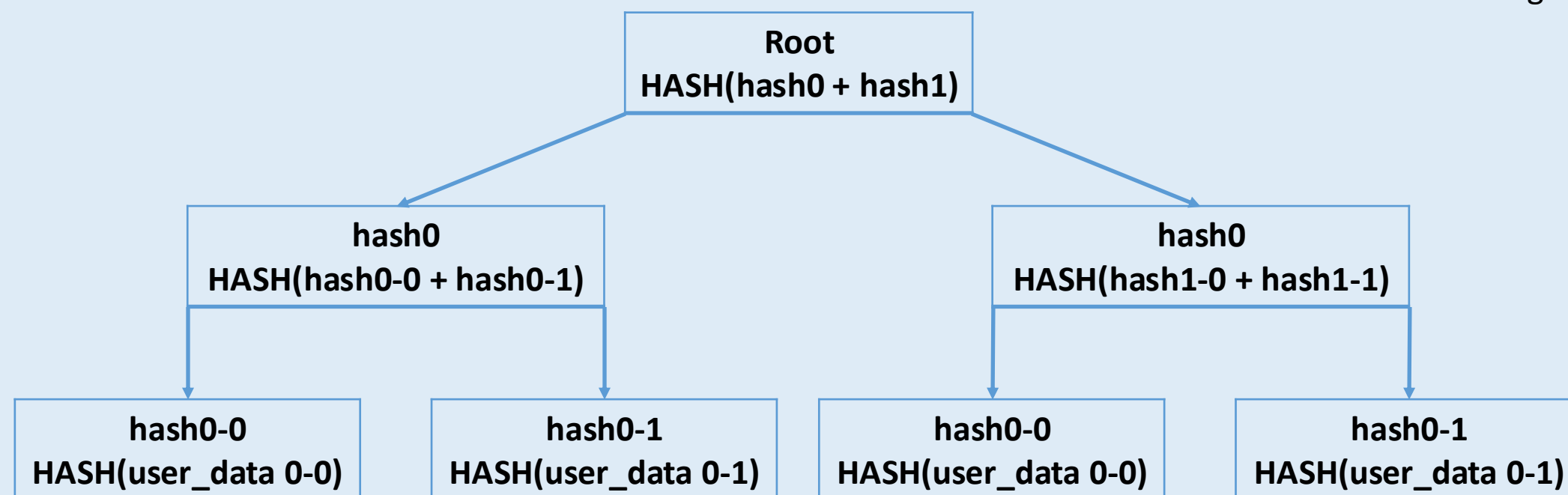
An example of consistency checks: Merkle trees

Remark: + means string concatenation.



An example of consistency checks: Merkle trees

Remark: + means string concatenation.



Applications:

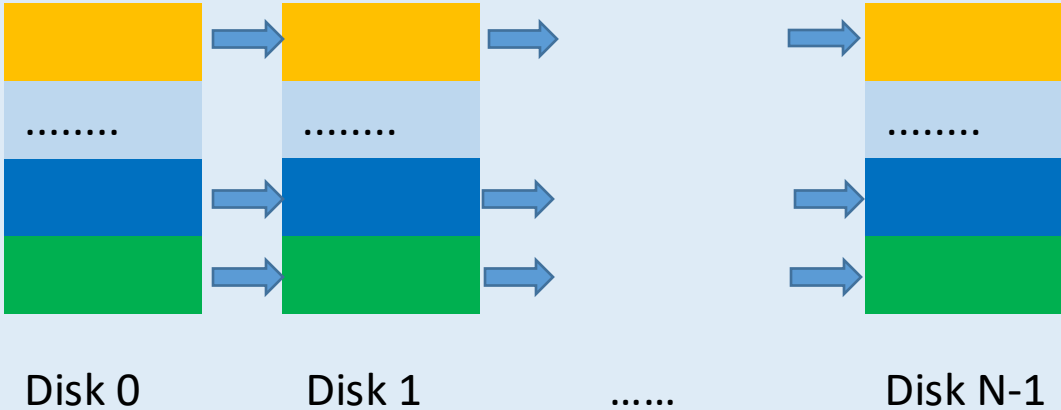
- ZFS organises the whole FS as a Merkle tree so it can check the consistency when walking a path and when looking for file extents,
- p2p networks use Merkle trees to ensure that the data is tamper-free,
- DynamoDB uses Merkle trees to quickly find parts of the DB that differ between cluster nodes and need to be resynced.

RAID – Redundant Array of Independent (Inexpensive) Disks

What can it achieve:

- Be more reliable than individual disks,
- Have bigger capacity than individual disks.

RAID levels

RAID0 (stipe)	<p>This RAID level splits data into parts that have $N * B$ bytes, then splits each part into N blocks, and writes blocks to individual disks:</p>  <p>The diagram illustrates the RAID 0 data distribution. It shows four vertical stacks of colored blocks representing data on different disks. The first stack is labeled 'Disk 0' and contains four blocks: yellow, light blue with '.....', blue, and green. The second stack is labeled 'Disk 1' and contains the same four blocks. Between the second and third stacks is an ellipsis '.....'. The fourth stack is labeled 'Disk N-1' and contains the same four blocks. Blue arrows indicate the flow of data from the first stack to the second, and from the second to the fourth, showing that data is striped across the disks.</p>
---------------	--

RAID levels

RAID1 (mirror)

Every disk contains the same data:

.....

Disk 0

==

==

.....

Disk 1

==

==

.....

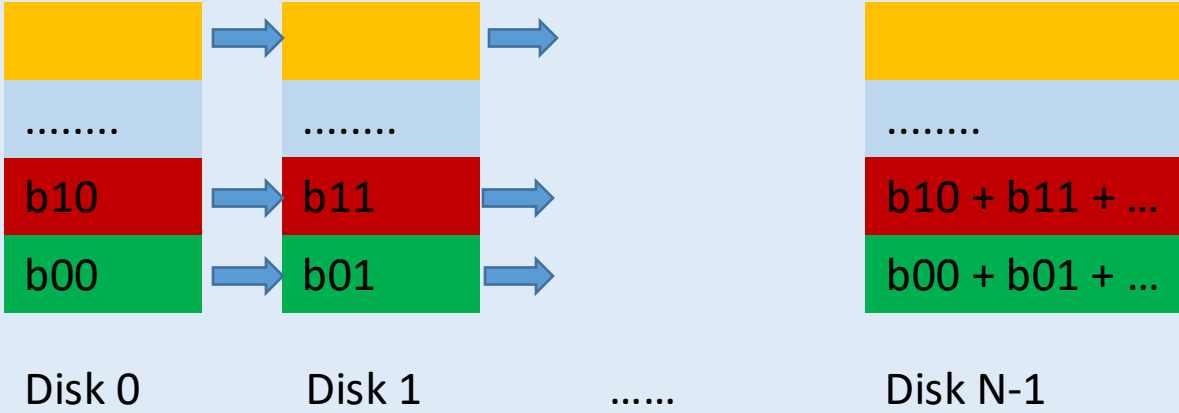
.....

Disk N-1

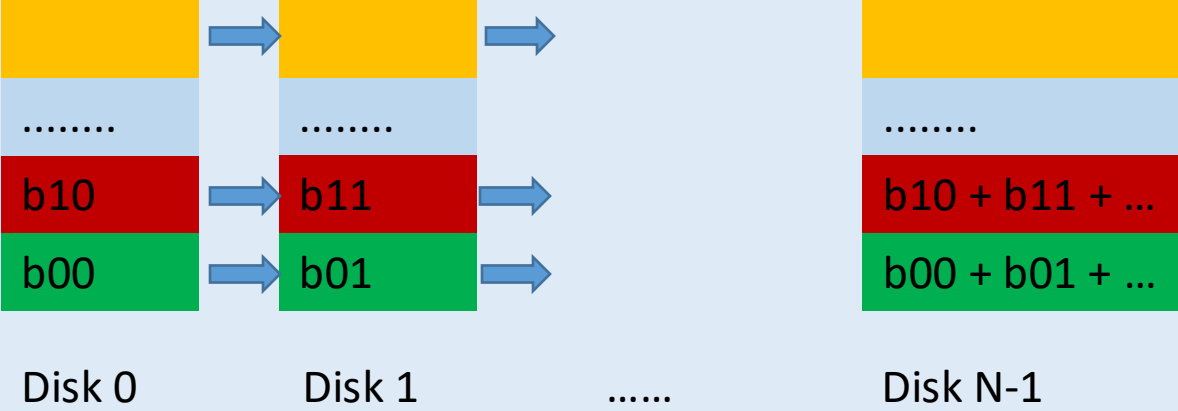
==

==

RAID levels

RAID4	<p>This RAID level uses N+1 disks. The first N disks are organised as a RAID0 array. Blocks on first N disks are XORed and the result is stored to the last disk.</p> <div><p>Disk 0 Disk 1 Disk N-1</p></div> <p>This RAID array may lose any disk and still retain all data.</p>
-------	---

RAID levels

RAID4	<p>This RAID level uses N+1 disks. The first N disks are organised as a RAID0 array. Blocks on first N disks are XORed and the result is stored to the last disk.</p>  <p>Diagram illustrating RAID 4 architecture:</p> <ul style="list-style-type: none">Disks: Disk 0, Disk 1, ..., Disk N-1, and a parity disk (Disk N).Data distribution: Data is striped across the first N disks (Disk 0 to Disk N-1) in a RAID0 configuration.Parity calculation: The parity for each stripe is calculated as the XOR (represented as addition in the diagram) of the data blocks on the first N disks.Example blocks: b_{00}, b_{01}, ..., b_{10}, b_{11} are shown on the data disks. The parity disk stores $b_{00} + b_{01} + \dots$ and $b_{10} + b_{11} + \dots$. <p>This RAID array may lose any disk and still retain all data.</p> <p>Such array has a disadvantage. The parity disk needs more IOPS than other member disks.</p>
-------	--

RAID levels

RAID5	<p>This RAID level works like RAID4, but places parity blocks on different disks in a round-robin manner:</p> <div><div><div></div><div>.....</div><div>b20</div><div>b10 + b11</div><div>b00</div></div><div><div></div><div>.....</div><div>b20 + b21</div><div>b10</div><div>b01</div></div><div><div></div><div>.....</div><div>b21</div><div>b11</div><div>b00 + b01</div></div></div>
-------	---

Write holes

Writes to different blocks in the same RAID stripe will happen at different moments.

Consider the following scenario:

1. a write to RAID1 starts,
2. member disk #0 processed the write,
3. the power failed,
4. member disk #1 has not processed the write.

Write holes

There is a hardware solution to write holes:

- BBU (Battery Backup Unit) in RAID controllers.

And there are software solutions:

- write intent bitmap (linux md),
- checksumming + COW (ZFS).

Write intent bitmap is useful in other ways. It both fixes write holes, and provides a way to decrease the time it takes to rebuild a RAID array.

One more fatal flaw of RAID5

Rebuilding a RAID5 array after a disk failure takes a long time*. Moreover, during that period the remaining disks are put under unusually large load. This increases the probability of a second disk failing during a rebuild.

As disks grew larger, this scenario went from theoretical to frequent. That is why we no longer use RAID5 and prefer RAID6 instead. RAID6 protects against failure of any 2 disks in an array.

** It takes approx. 1.5 day to overwrite a 10Tb disk at 100Mb/sec.*

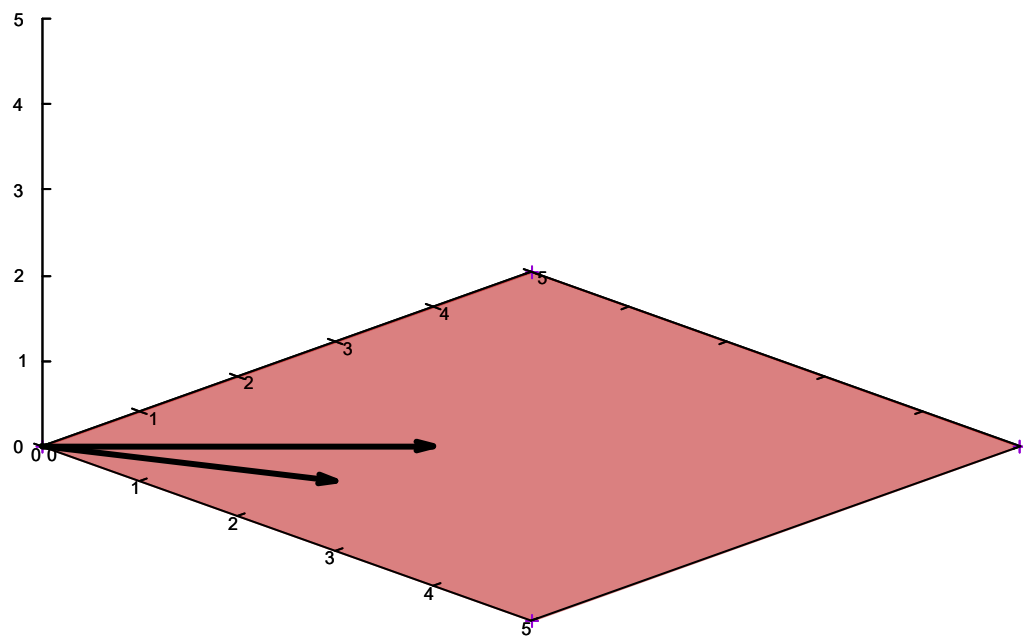
RAID6 and Reed-Solomon codes

RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.

RAID6 and Reed-Solomon codes

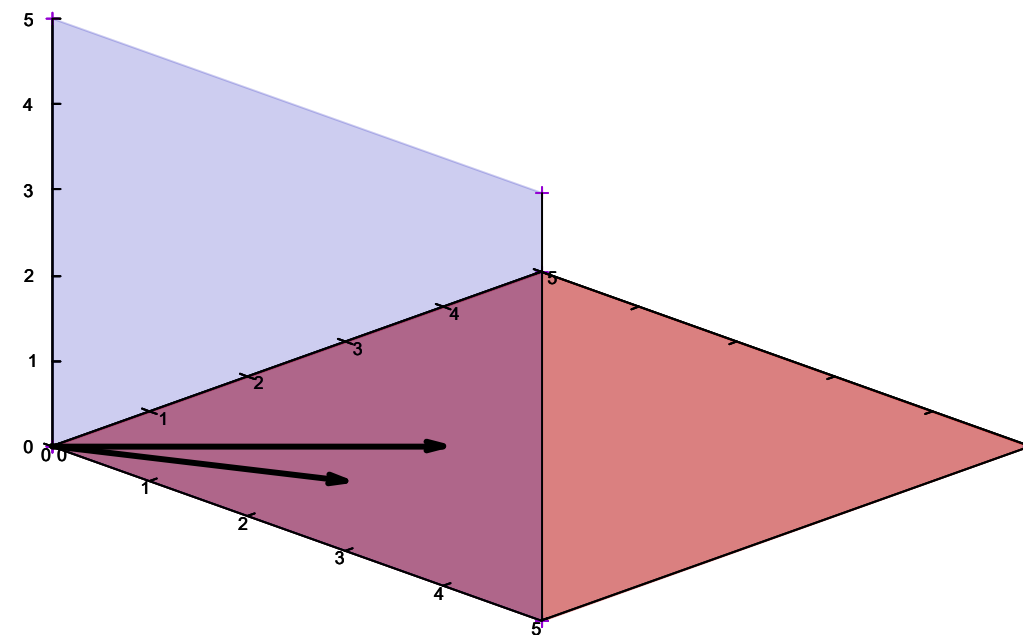
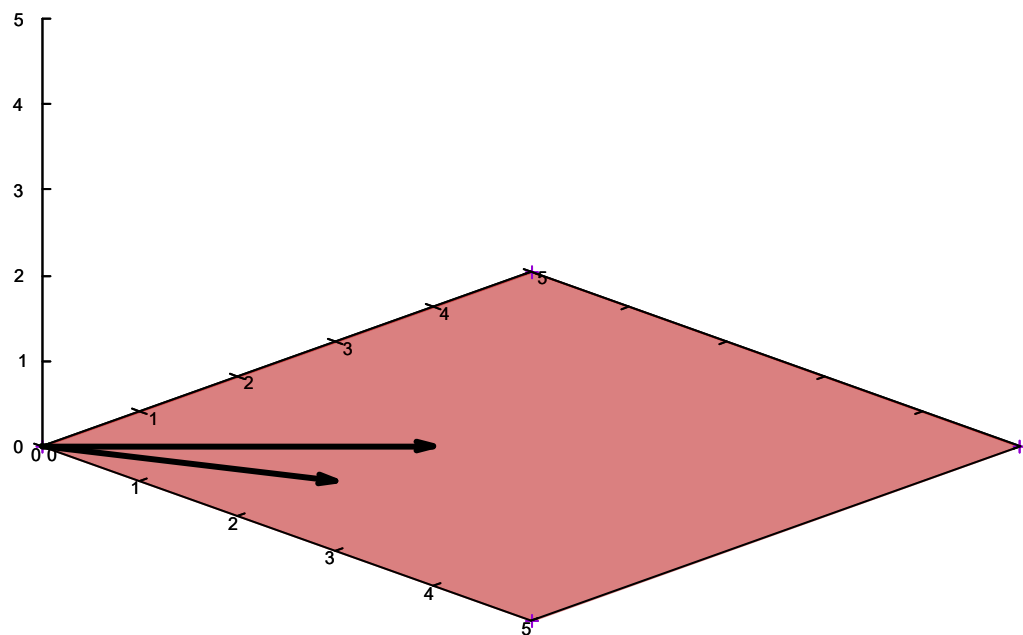
A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.



RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.

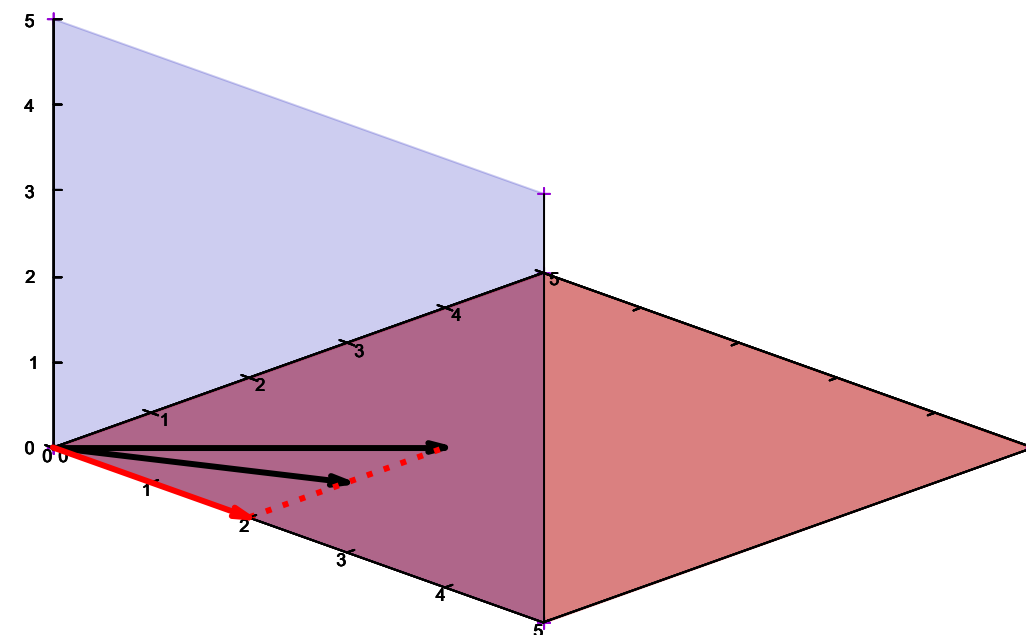
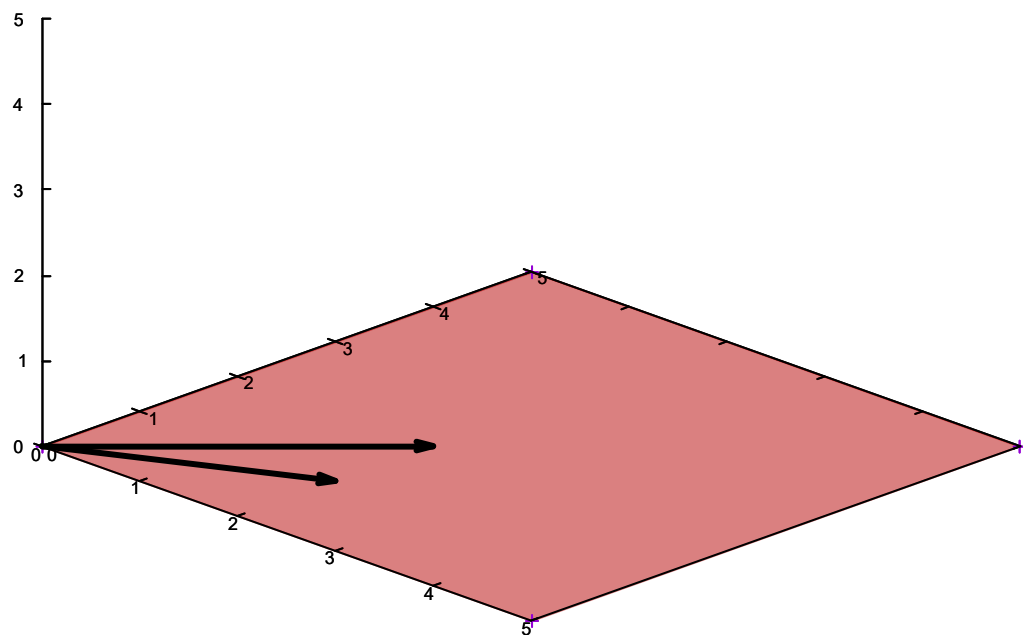
Losing a disk that stores the "y-coordinate of the image" may be modeled as projecting the vector to the plane $\{y = 0\}$.



RAID6 and Reed-Solomon codes

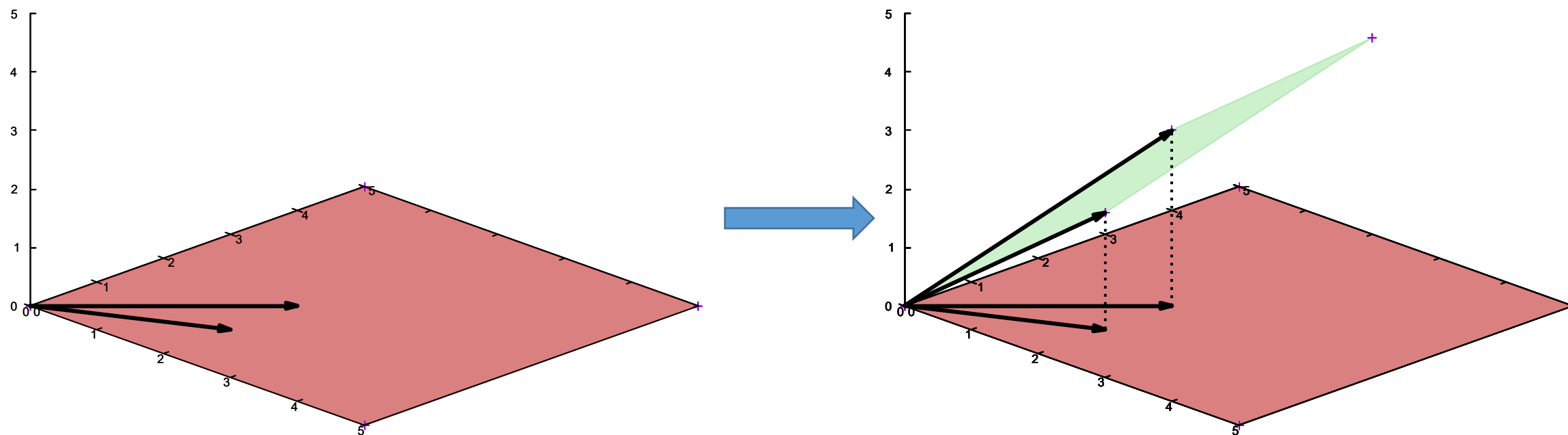
A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.

Losing a disk that stores the "y-coordinate of the image" may be modeled as projecting the vector to the plane $\{y = 0\}$.



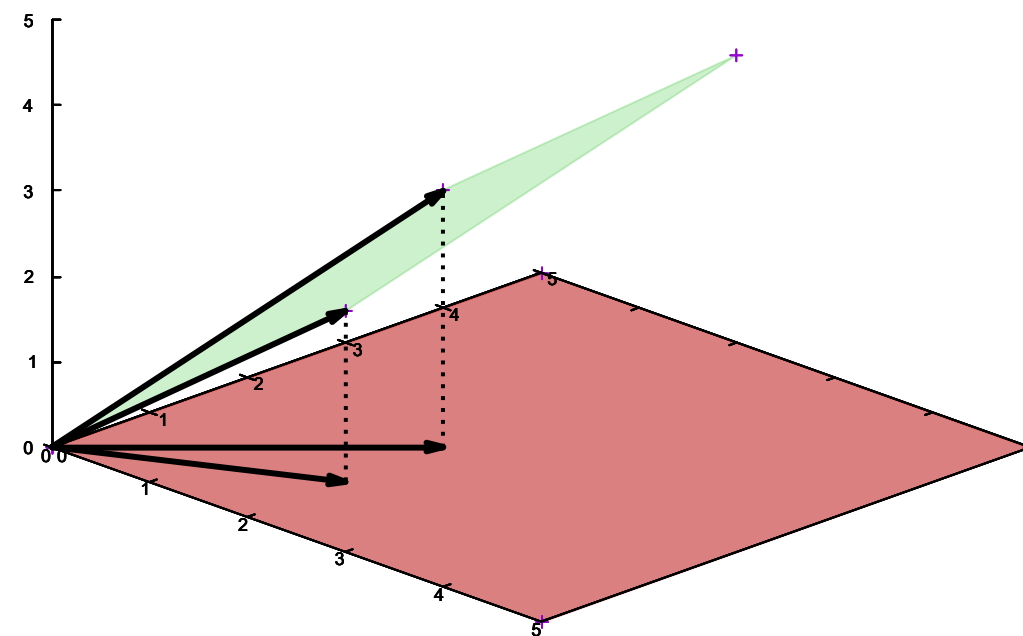
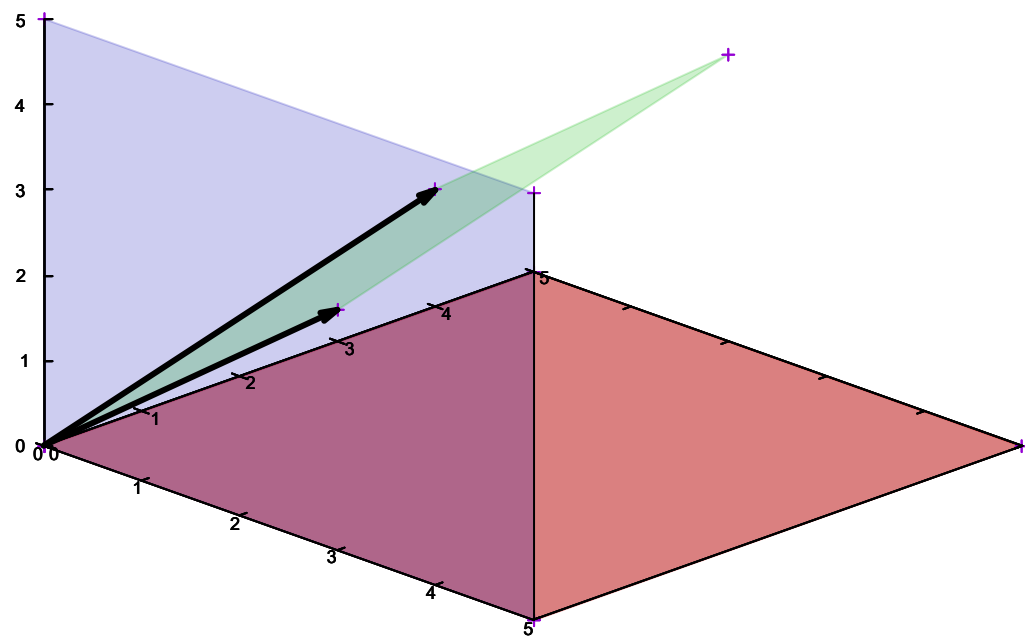
RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.



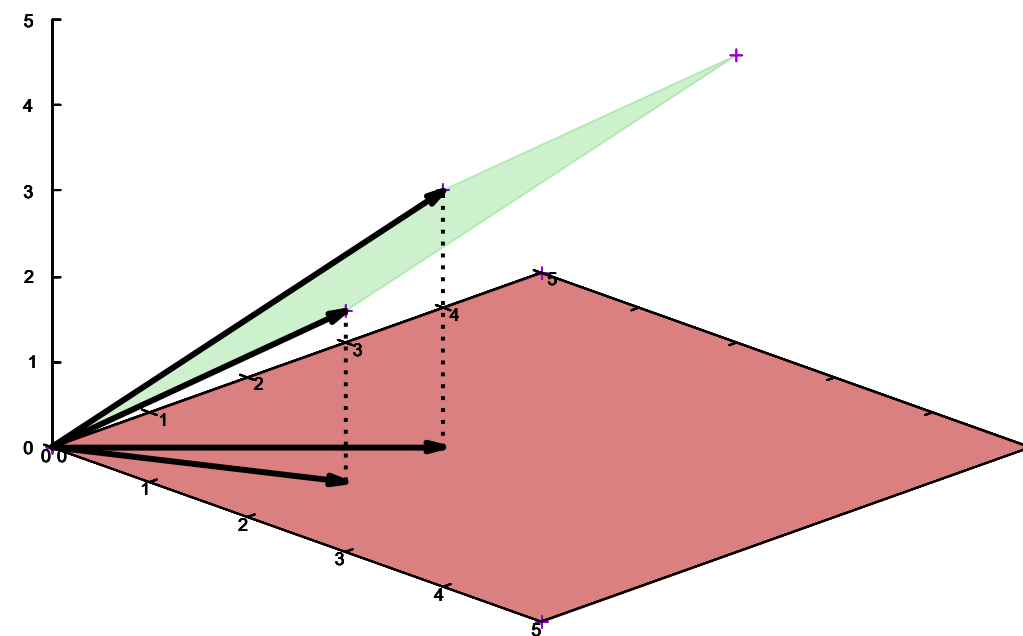
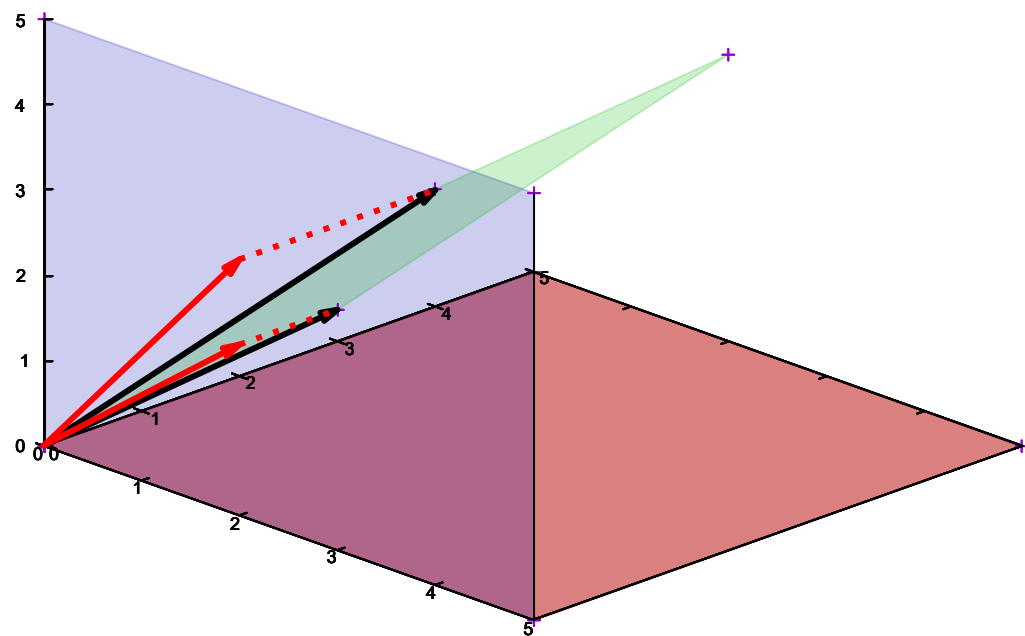
RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.



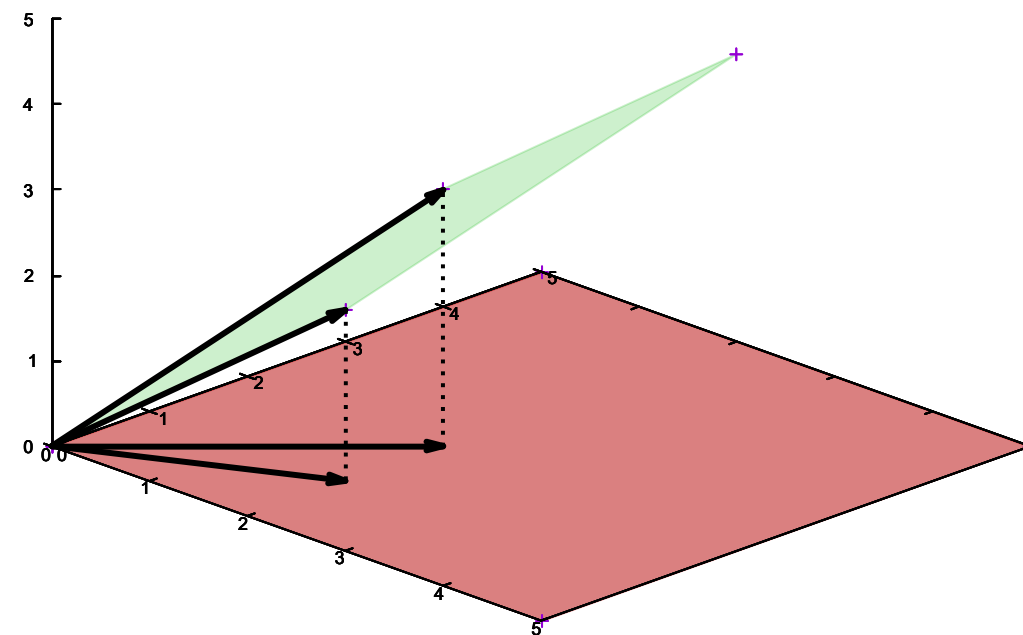
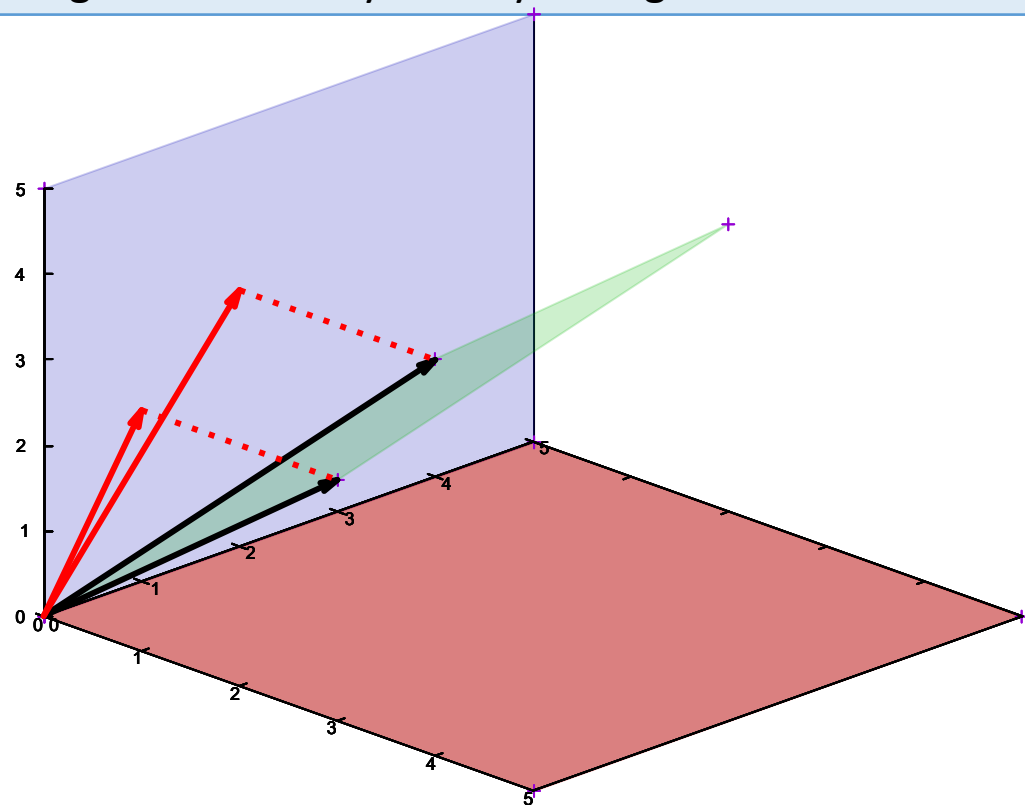
RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.



RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.



RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.

RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.

Let us choose an embedding $\varphi: k^n \hookrightarrow k^{n+m}$ such that its image is transversal to every m -dimensional coordinate plane.

Replacing a message x that has n bytes with $\varphi(x)$ gives a message that has $n + m$ bytes and has the following additional property: x can be uniquely reconstructed from any n coordinates of $\varphi(x)$.

RAID6 and Reed-Solomon codes

A message that has n bytes may be regarded as a vector in the n -dimensional vector space over the field $k = GF(256)$.

Let us choose an embedding $\varphi: k^n \hookrightarrow k^{n+m}$ such that its image is transversal to every m -dimensional coordinate plane.

Replacing a message x that has n bytes with $\varphi(x)$ gives a message that has $n + m$ bytes and has the following additional property: x can be uniquely reconstructed from any n coordinates of $\varphi(x)$.

Applications choose φ in such a way that the first n coordinates of $\varphi(x)$ coincide with x .

RAID6 and Reed-Solomon codes

Let us choose an embedding $\varphi: k^n \hookrightarrow k^{n+m}$ such that its image is transversal to every m -dimensional coordinate plane.

Replacing a message x that has n bytes with $\varphi(x)$ gives a message that has $n + m$ bytes and has the following additional property: x can be uniquely reconstructed from any n coordinates of $\varphi(x)$.

How do we prove that such embeddings $\varphi: k^n \hookrightarrow k^{n+m}$ exist?

1. Let k be the field with q elements. Show that the number of collections of n linearly independent vectors in k^{n+m} is
$$(q^{n+m} - 1) \cdot (q^{n+m} - q) \cdot \dots \cdot (q^{n+m} - q^{n-1})$$
2. Let $GL_k(n)$ be the group of invertible $n \times n$ matrices with coefficients in k . Show that the number of elements in $GL_k(n)$ is
$$(q^n - 1) \cdot (q^n - q) \cdot \dots \cdot (q^n - q^{n-1})$$
3. Show that the number of n -dimensional subspaces in k^{m+n} is
$$s(n, n + m) = \frac{(q^{n+m} - 1) \cdot (q^{n+m} - q) \cdot \dots \cdot (q^{n+m} - q^{n-1})}{(q^n - 1) \cdot (q^n - q) \cdot \dots \cdot (q^n - q^{n-1})}$$
4. Show that the number of n -dimensional subspaces in k^{m+n} that contain at least one coordinate axis is much smaller than $s(n, n + m)$.

Problems of RAID6

Let us use Reed-Solomon coding with parameters $(n, n + m)$.
This means that we store m redundant pages for every n pages
of user data.

Problems of RAID6

Let us use Reed-Solomon coding with parameters $(n, n + m)$.
This means that we store m redundant pages for every n pages of user data.

1. Tail latency:
 - In order to read n consecutive data pages, we need to make requests to n disks or n cluster nodes.

Problems of RAID6

Let us use Reed-Solomon coding with parameters $(n, n + m)$.
This means that we store m redundant pages for every n pages of user data.

1. Tail latency:

- In order to read n consecutive data pages, we need to make requests to n disks or n cluster nodes.
- We can use longer pages. This way a bigger share of requests may be satisfied by reading only one disk.
- We can read redundant disks as well and treat slow disks as if they failed.

Problems of RAID6

Let us use Reed-Solomon coding with parameters $(n, n + m)$.
This means that we store m redundant pages for every n pages of user data.

1. Tail latency:

- In order to read n consecutive data pages, we need to make requests to n disks or n cluster nodes.
- A single write to an array requires writes to $n + m$ disks.

Problems of RAID6

Let us use Reed-Solomon coding with parameters $(n, n + m)$. This means that we store m redundant pages for every n pages of user data.

1. Tail latency:

- In order to read n consecutive data pages, we need to make requests to n disks or n cluster nodes.
- A single write to an array requires writes to $n + m$ disks.
- We can use Reed-Solomon with parameters $(n, n + m + k)$ and wait only for $n + m$ disks to confirm writes. Remaining disks may be assumed failed.

Problems of RAID6

Let us use Reed-Solomon coding with parameters $(n, n + m)$.
This means that we store m redundant pages for every n pages of user data.

1. Tail latency:
 - In order to read n consecutive data pages, we need to make requests to n disks or n cluster nodes.
 - A single write to an array requires writes to $n + m$ disks.
2. Read amplification during a rebuild:
 - In order to rebuild 1 page, we must read n pages.

Problems of RAID6

Let us use Reed-Solomon coding with parameters $(n, n + m)$. This means that we store m redundant pages for every n pages of user data.

1. Tail latency:

- In order to read n consecutive data pages, we need to make requests to n disks or n cluster nodes.
- A single write to an array requires writes to $n + m$ disks.

2. Read amplification during a rebuild:

- In order to rebuild 1 page, we must read n pages.

- There is a class of erasure codes that minimises the replication traffic: Minimum Storage Regenerating codes. See: Clay Codes: Moulding MDS Codes to Yield an MSR Code <https://www.usenix.org/system/files/conference/fast18/fast18-vajha.pdf>

The model of errors in RAID6

Reminder: RAID6, just like RAID5 assumes that a failed disk is reported as missing. RAID6 provides no protection against the **bit rot**. It does not detect situations when a failed disk reads garbage and reports success.

If we use RAID6 we still need data integrity checks.