Paper 008-2010

The French Connection – Implementing a RAID Controller Algorithm in Base SAS®

Koen Vyverman, SAS Institute B.V., The Netherlands
Paul M. Dorfman, Independent SAS® Consultant, Florida

ABSTRACT

RAID configuration of storage arrays is fairly ubiquitous nowadays, and many IT professionals have a conceptual awareness of how RAID provides redundancy and thus protects against data loss when one or more disks in an array should fail. However, few people know exactly how this redundancy works on the level of bits and bytes... In this paper we reveal the inner workings of RAID arrays, and discover that the theoretical underpinnings lean heavily upon the work of a 19th century flamboyant French algebraist. Eschewing mathematical formalism, we explain concepts and highlight them with simple examples. As a practical application, we show how a RAID controller algorithm can be implemented in Base SAS®.

INTRODUCTION

Disk arrays are everywhere nowadays: we find them purring away in the subterranean server-rooms of every sizable company, stacked in the racks of data centers where they offer petabytes of on-line storage in Storage Area Networks (SAN), and even in private homes where they serve multimedia content for the household as network-attached storage devices (NAS).

A disk array is in essence nothing more than an enclosure holding a number of individual hard disks, and a controller chip to handle multiple parallel I/O streams to and from those disks. Even though there may be up to 15 single disks in an array — typical for rack-mounted systems — the disk array controller can make it look to the outside world as if there is just one large storage volume. So if you stick 15 relatively cheap 1TB hard disks into a disk array, clients connecting to it will see a whopping 15TB data volume!

This promise of massive storage capacity, together with the I/O performance gain caused by writing (or reading) files to (or from) a number of disks simultaneously, is what makes the use of disk arrays so attractive. Indeed, when a file is written to the array, the controller chip will chop it up and deposit parts of the file onto all available disks.

There is one major problem though with such a system: the second law of thermodynamics states that entropy must increase. In other words: if you wait long enough, just about anything will eventually break down. And so do hard disks. If every file on a disk array is smeared out over any number of disks, it follows that if one single disk in the array goes belly up, the loss of every file on the array ensues because parts will be missing. As a preventive measure against data loss, it is common practice to make regular back-ups of the entire array onto tapes, allowing a full restore in the event of disaster.

However, companies and data centers prefer to have their data storage on-line and accessible at all times, even when one of the disks crashes. How is this done? In the late eighties, a technology named RAID was first developed. RAID is an acronym for Redundant Array of Inexpensive Disks, and it describes mechanisms to introduce redundancy in the operation of disk arrays and thus improve reliability by providing a certain resilience to hard disk crashes. In the decades since RAID first emerged, a number of variations on the original idea were developed. These are known as RAID schemes or RAID levels, numbered 0 through 6, and some of them have become a de facto industry standard wherever disk arrays are deployed.

The funny thing however is that although most IT professionals responsible for corporate SAN or NAS systems have a conceptual understanding of how RAID works, very few really know exactly how RAID is able to provide redundancy on the level of bits and bytes. The main purpose of this paper is to explain the inner workings of RAID redundancy, and as a practical example we show how a RAID algorithm might be implemented in Base SAS.

We first devote a section to a general overview of the standard RAID levels, and subsequently describe in more detail how RAID-4, RAID-5, and RAID-6 are defined. It soon becomes apparent that in order to attain the promised RAID redundancy, we must find a way to perform computations on bytes, involving a way in which to add and multiply them in some meaningful manner. In the next section we explore two naive approaches to adding and multiplying bytes, and conclude that things are not as simple as that! We then proceed to introduce the French Connection in the shape of the ideas of a 19th century French mathematician, which eventually lead to a solution for our conundrum and tell us how to add and multiply bytes. In the final section of the paper we present Base SAS code to demonstrate the workings of a RAID algorithm by writing a file to a mock-up disk array, simulating a hard disk crash by deleting the contents of a disk, and subsequently rebuilding the missing parts allowing us to resurrect our file.

STANDARD RAID LEVELS

In their seminal paper “A Case for Redundant Arrays of Inexpensive Disks” (Patterson et al., 1988) Patterson, Gibson, and Katz presented the concept of five different RAID levels which would later crystallize as RAID-1 through RAID-5. Since then, various other flavours and combinations of RAID mechanisms have been added to the list, and the prominent SAN and NAS vendors even tend to have their own proprietary algorithms built into the RAID controllers of their disk arrays.

Some of the RAID levels were conceived specifically to improve the reliability of storage arrays by introducing redundancy in such manner that failure of one disk in the array will not result in the loss of all data on the entire array. Other RAID schemes focus on boosting I/O performance by reading (writing) chunks of a file in parallel from (to) many disks at once. As is frequently the case, there is a trade-off between the two, and depending on requirements a careful choice must be made which RAID level to implement.

Three mechanisms are used in RAID level specification: mirroring, striping, and redundancy (also referred to as ‘parity’):

- Mirroring writes the same data to multiple physical disks in the array. Obviously this provides a high level of redundancy, but it also comes at a cost since 50% or more of the total storage capacity is used for duplicate data.
- Striping is the mechanism whereby a single file is split into chunks which are stored on multiple physical disks, and as such it is the performance boosting technique in the RAID toolkit.
- Redundancy pertains to the computation of new data (parity blocks) from striped data, and storing it on one or more physical disks. The parity data is computed in such a way that it can be used to reconstruct missing stripes of data in the event of the loss of a disk in the array.

The following list shows the standard RAID levels and their main characteristics:

- RAID-0: Improved performance by striping across at least two disks. Provides no redundancy.
- RAID-1: Simple mirroring of disks. High redundancy factor, but low storage efficiency since each disk in the array contains the same information.
- RAID-2: Micro-striping with single bytes written across the disks in the array. Error correction codes and parity data are calculated and stored on multiple parity disks.
- RAID-3: Striping with byte-level redundancy data stored on a dedicated disk in the array.
- RAID-4: Striping with block-level (as in: file-system block) redundancy information stored on a dedicated disk in the array.
- RAID-5: Striping with block-level distributed redundancy data. For each stripe a parity block is computed. Parity blocks are not stored on a dedicated disk, but are distributed in cyclic fashion over all the disks in the array. Provides single drive failure protection.
- RAID-6: Striping with block-level dual distributed parity data. Two parity blocks are computed for each stripe of data-blocks. Parity blocks are distributed in cyclic manner over all disks in the array. Provides protection against up to two disk failures.

For the remainder of the paper we will focus on the latter three RAID levels. RAID-5 is probably the most common one. RAID-4 and RAID-6 are conceptually very similar to RAID-5.

BLOCK-LEVEL RAID SCHEMES

In this section we conceptually discuss the block-level RAID schemes: RAID-4, RAID-5, and RAID-6. The basic concept for all three varieties is that parity blocks (i.e. redundancy information) are computed from the blocks of actual data which are being written to the array. We will get to the details of this computation shortly, but first we discuss the similarities and differences between the three RAID levels.

RAID-4

As briefly introduced in the previous section, RAID-4 uses block-level striping and stores a computed parity-block on a dedicated disk. Consider the following generic set-up: assume we have a disk array consisting of N physical disks of the same capacity, with the same type of formatting, and the same block-size. If this array is RAID-4 configured, then writing a file to it entails the writing of a number of stripes, each of which contains N-1 data-blocks with a size equal to the block-size, and a parity block also of block-size. The stripes’ data-blocks are spread out over the first N-1 array disks, and the parity blocks are collected on the one remaining disk. The parity blocks are constructed in such manner that when one of the disks in the array dies, the missing block in each stripe can be reconstructed. As such, RAID-4 protects against the loss of one disk. Table 1 shows the distribution of stripes, data-blocks, and parity blocks across a 5-disk RAID-4 array.

	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
Stripe 1	data$_{1,1}$	data$_{1,2}$	data$_{1,3}$	data$_{1,4}$	parity$_1$
Stripe 2	data$_{2,1}$	data$_{2,2}$	data$_{2,3}$	data$_{2,4}$	parity$_2$
Stripe 3	data$_{3,1}$	data$_{3,2}$	data$_{3,3}$	data$_{3,4}$	parity$_3$
Stripe 4	data$_{4,1}$	data$_{4,2}$	data$_{4,3}$	data$_{4,4}$	parity$_4$
Stripe 5	...	...	...	...	...

Table 1. Distribution of data and parity blocks on a 5-disk RAID-4 array.

Suppose that disaster strikes: Disk 2 crashes and is replaced by a newly formatted disk. Thanks to the parity blocks possessing this magical quality of warranting redundancy, the contents of Disk 2 can be re-built by the RAID controller chip by performing some operation on the remaining pieces of the stored files, and the parity blocks on Disk 5. If it is Disk 5 which must be replaced, then the RAID controller merely needs to re-compute all the parity blocks, based upon the pieces of files in each stripe.

RAID-5

The only difference between RAID-4 and RAID-5 is that in the latter scheme, the parity blocks are not stored separately from the data-blocks. Instead, they are interleaved with the data-blocks on all disks in the array. Various schemes exist for picking the disk where the parity block for a given stripe is written. The simplest schemes are cyclic, rotating either left to right through the array, or right to left. With one parity block per stripe, RAID-5 protects against the loss of a single disk, just like RAID-4. Table 2 shows a possible distribution of stripes, data-blocks, and parity blocks across a 5-disk RAID-5 array with left-to-right parity block interleaving.

	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
Stripe 1	data$_{1,1}$	data$_{1,2}$	data$_{1,3}$	data$_{1,4}$	parity$_1$
Stripe 2	parity$_2$	data$_{2,1}$	data$_{2,2}$	data$_{2,3}$	data$_{2,4}$
Stripe 3	data$_{3,1}$	parity$_3$	data$_{3,2}$	data$_{3,3}$	data$_{3,4}$
Stripe 4	data$_{4,1}$	data$_{4,2}$	parity$_4$	data$_{4,3}$	data$_{4,4}$
Stripe 5	...	...	...	...	...

Table 2. Distribution of data and parity blocks on a 5-disk RAID-5 array.

Although RAID-5 seems only a variation on RAID-4, it is by far the more popular of the two. The reason for this is that in a RAID-4 disk array, the dedicated parity disk may easily become an I/O bottleneck when many file-updates take place. Indeed, every time a file is modified some of its data-blocks will need to be re-written and the parity block for each affected stripe needs to be re-computed. Since all RAID-4 parity blocks reside on the same disk, pseudo-random read/write operations are needed on the parity disk and the drive head is sent skittering all over the place to update a parity block here, another one there, thus dragging down the performance of the entire array. Furthermore, if certain files (like RDBMS tables) are constantly being updated, there is a risk of creating hotspots in the area where the file's parity blocks are located, which may lead to data corruption and eventually disk failure. In summary: it is considered a good thing if the wear and tear on every disk in the array is as uniform as possible. Hence the preference for RAID-5 rather than RAID-4.

RAID-6

The RAID-6 scheme looks similar to RAID-5, with the notable exception that for each stripe not one but two parity blocks are computed and written to different disks in the array. Table 3 shows a possible distribution of stripes, data-blocks, and parity blocks across a 5-disk RAID-6 array with left-to-right parity block interleaving.

By storing two differently computed parity blocks, RAID-6 offers protection for up to two disks in the array kicking the bucket. Indeed, when reconstructing the missing blocks in a stripe after replacing two disks, the following may be the case:

- All data-blocks are intact, but both parity blocks are missing: the RAID controller then simply re-computes both parity blocks.

- Two of the data-blocks are missing: reconstruct them by using the single remaining data block and both parity blocks.
- One of the data-blocks and one of the parity blocks are gone: first reconstruct the missing data-block by means of the remaining two data-blocks and the remaining parity block. Then re-compute the missing parity block by means of the three data-blocks.

	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
Stripe 1	data$_{1,1}$	data$_{1,2}$	data$_{1,3}$	parity$_{1,A}$	parity$_{1,B}$
Stripe 2	parity$_{2,B}$	data$_{2,1}$	data$_{2,2}$	data$_{2,3}$	parity$_{2,A}$
Stripe 3	parity$_{3,A}$	parity$_{3,B}$	data$_{3,1}$	data$_{3,2}$	data$_{3,3}$
Stripe 4	data$_{4,1}$	parity$_{4,A}$	parity$_{4,B}$	data$_{4,2}$	data$_{4,3}$
Stripe 5	...	...	...	...	...

Table 3. Distribution of data and parity blocks on a 5-disk RAID-6 array.

All very well, but what exactly are these mysterious parity blocks, how are they constructed from the data blocks, and why do they have this magical property of ensuring redundancy?

The construction and computation of the parity blocks is a result from a branch of mathematics known as Coding Theory. One of the main areas of application of Coding Theory is the development of algorithms to boost the robustness of data transmissions via particular channels, such as radio waves, telephony, and indeed, digital data being written to — or read from — hard disks. Coding algorithms (or 'codes') will typically add extra data to a signal before transmission, while decoding algorithms will use these extra bits for error correction to restore any parts of the signal that may have been lost or otherwise compromised between the transmitter and the receiver.

Any kind of foray into Coding Theory territory would lead us too far astray however, so we shall forego the details. The avid reader may wish to peruse the papers by Plank (1997) and Anvin (2009) for more information. Especially the former is a rather accessible tutorial.

For now, it suffices to envision the parity block calculation as some kind of mathematical formula into which we stick the data-blocks of a stripe, and it will evaluate to the parity block for the stripe. As such, the computation of the parity blocks involves performing mathematical operations on a bunch of bytes. It is important to stress that these operations are done on the actual bytes, and not on numbers or strings, or whatever else we may perceive those bytes to represent!

The formula needed to compute the parity block for the RAID-4 and RAID-5 scheme as well as one of the two RAID-6 parity blocks turns out to be fairly straightforward. It is simply the 'sum' of all the bytes in the data-blocks of a given stripe. The formula for the second RAID-6 parity block is significantly more complex and also involves 'multiplying' bytes with one another. As a matter of fact, the Coding Theory algorithm to compute the second RAID-6 parity block uses something called a Reed-Solomon Code. Reed-Solomon Codes are also used for error correction in audio CD's, and contain an amount of redundancy data to allow reconstruction of the audio signal in the event of scratches on the disk surface.

Take heed of the quoted words in the above: it yet remains to be seen what exactly it means to 'add' or 'multiply' two bytes. In the next section we will begin to explore how to do arithmetic with bytes.

THE NAIVE APPROACH

Let us first inspect the objects we are dealing with here, the set of 8-bit bytes. Table 4 enumerates all such bytes. We number each byte for easy reference, and we also show some formal representations of bytes: sequences of bits, 8-tuples, the classical interpretation as integers, or even a bunch of polynomials.

Note that we have chosen the byte numbering to run from 0 to 255 rather than from 1 to 256. This proves to be more elegant, since the byte number now coincides with the integer representation. The polynomial representation may seem a bit odd at first, but it is a perfectly legitimate formalism to represent the set of 256 8-bit bytes as the set of 256 7th-degree polynomials with coefficients either 0 or 1. The usefulness of the polynomial representation will become apparent further on.

Byte Number	Bit-string Representation	8-tuple Representation	Integer Representation	Polynomial Representation
0	00000000	(0,0,0,0,0,0,0,0)	0	0
1	00000001	(0,0,0,0,0,0,0,1)	1	1
2	00000010	(0,0,0,0,0,0,1,0)	2	x
3	00000011	(0,0,0,0,0,0,1,1)	3	$x+1$
4	00000100	(0,0,0,0,0,1,0,0)	4	x^2
5	00000101	(0,0,0,0,0,1,0,1)	5	x^2+1
6	00000110	(0,0,0,0,0,1,1,0)	6	x^2+x
...	...	...	...	...
254	11111110	(1,1,1,1,1,1,1,0)	254	$x^7+x^6+x^5+x^4+x^3+x^2+x$
255	11111111	(1,1,1,1,1,1,1,1)	255	$x^7+x^6+x^5+x^4+x^3+x^2+x+1$

Table 4. List of 8-bit bytes and various ways to represent them.

Computing something means: to perform arithmetical operations on objects. Arithmetical operations are addition, subtraction, multiplication, and division. We can focus on addition and multiplication. If we have a good addition and a good multiplication, then the other operations follow automatically because subtraction is nothing but inverse addition, and division is inverse multiplication. Once we figure out how to add and multiply the bytes in our set, we can also subtract and divide them.

Notation convention: in order to distinguish between the addition and multiplication of integers as we all know them and the same operations on bytes, we will henceforth use the following symbols:

$+$	Addition on integers	$\oplus$	Addition on bytes
x	Multiplication on integers	$\otimes$	Multiplication on bytes

Table 5. Operator naming convention.

So what are the requirements for a good addition and a good multiplication?

- The addition must have a zero-element in the set: there must be a specific byte B_0 such that if we add it to any byte B_i, the result remains byte B_i. Symbolically, for any B_i: $B_i \oplus B_0 = B_i$.
- The multiplication must have a unit-element in the set: there must be a specific byte B_1 such that if we multiply it with any byte B_i, the result remains B_i. Symbolically, for any B_i: $B_1 \otimes B_i = B_i$.
- Every element in the set must have an additive inverse: for every byte B_i there must be some byte B_j such that the sum of the two is the zero-element B_0. Symbolically, for any B_i there is a B_j satisfying $B_i \oplus B_j = B_0$.
- Every element in the set, except the zero-element for the addition, must have a multiplicative inverse: for every byte B_i with $i \neq 0$ there is some byte B_j such that the product of the two is the unit-element B_1. Symbolically, for any B_i with $i \neq 0$ there is a B_j satisfying $B_i \otimes B_j = B_1$.
- The addition and the multiplication must be commutative: for any two bytes B_i and B_j, it must be so that $B_i \oplus B_j = B_j \oplus B_i$ and $B_i \otimes B_j = B_j \otimes B_i$.
- The addition and the multiplication must be associative. For any three bytes B_i, B_j, and B_k, it must be so that $B_i \oplus (B_j \oplus B_k) = (B_i \oplus B_j) \oplus B_k$ and $B_i \otimes (B_j \otimes B_k) = (B_i \otimes B_j) \otimes B_k$.
- The multiplication must be distributive over the addition. For any three bytes B_i, B_j, and B_k, it must be so that brackets can be worked out: $B_i \otimes (B_j \oplus B_k) = (B_i \otimes B_j) \oplus (B_i \otimes B_k)$.
- And last but not least, for any two bytes, both their sum and their product must be defined within the set of 256 bytes.

Looking at the integer representation in Table 4, it might be tempting to try defining the addition on bytes as the sum of their corresponding integers. E.g. $B_2 \oplus B_4$ would then evaluate to 2 + 4 = 6, which corresponds to B_6. This is rather naive of course, and it only works for the lower bytes in our set of 256. We cannot add B_{200} and B_{100} in this manner, because there is no B_{300} in the set of 8-bit bytes. Going by the necessary properties listed above, this cannot be a good addition.

How about something more clever? Since we work in a finite set, we can try a modular addition. It feels rather natural to wrap back to the beginning of the set once we run out of bytes at the end. So let's try that. We define $B_i \oplus B_j = B_{(i+j) \bmod 256}$ and $B_i \otimes B_j = B_{(ixj) \bmod 256}$. Will this yield a good addition and multiplication? Rather hard to see on a set of size 256, so let's check with a smaller number of bytes whether the idea is viable.

Suppose we work with the set of 2-bit bytes as listed in Table 6. Since there are only 4 elements, we can quickly derive the full addition and multiplication tables.

Byte	Bit-string	Integer
B_0	00	0
B_1	01	1
B_2	10	2
B_3	11	3

Table 6. The full set of 2-bit bytes.

The modular addition and multiplication definitions now become: $B_i \oplus B_j = B_{(i+j) \bmod 4}$ and $B_i \otimes B_j = B_{(ixj) \bmod 4}$. Example: $B_3 \oplus B_2 = B_{(3+2) \bmod 4} = B_{5 \bmod 4} = B_1$. Or a multiplication: $B_3 \otimes B_2 = B_{(3x2) \bmod 4} = B_{6 \bmod 4} = B_2$. The resulting addition and multiplication tables are shown in Table 7 and 8 respectively. Only the byte numbers are printed in the tables, which makes the structure much more transparent.

⊕	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Table 7. The modular addition on the set of of 2-bit bytes.

⊗	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Table 8. The modular multiplication on the set of of 2-bit bytes.

Looking at the addition table, there's nothing obviously wrong with it. Indeed, the byte B_0 appears to function as a zero-element, and the table is nicely symmetrical which reflects the commutative property of ⊕. Every byte also has an additive inverse. Then let's inspect the multiplication table: B_1 works nicely as a unit-element for the multiplication. The table is symmetrical, which proves the commutativity of ⊗. But alas! The second byte B_2 has no multiplicative inverse! There is no other byte which when multiplied with B_2 yields B_1.

We must perforce conclude that modular arithmetic on the integer representation of bytes does not enable us to define an addition and multiplication on a set of bytes either. So how is it done? We could keep trying of course until we stumble on a definition which satisfies all of the conditions (*supra*), but since we are looking for a good definition of $\oplus$ and $\otimes$ for a set of 256 bytes, this would quickly become a rather tedious exercise. Luckily, it has all been done before. Longtime before, even...

THE FRENCH CONNECTION

During the first decades of the 19th century, with the French Revolution which saw the abolishment of centuries of monarchy still fresh in the memories of its people, France was a country in a state of political turmoil. Efforts to remodel the government as a republic had been short-lived, and were cut short with the rise of Napoleon Bonaparte as self-styled Emperor in 1804. The Napoleonic era was in its turn followed by a return to monarchy under king Charles X, which ended in another revolution (the July Revolution of 1830) after which royalists and republicans kept bickering big-time for control of the country in a new setting of constitutional monarchy.

It is amidst all this upheaval that on October 25th 1811, Évariste Galois is born in Bourg-la-Reine, a township just outside of the Paris city walls, now long since engulfed by the expanding metropolis. At age 12, the young Galois entered the Lycée Louis-le-Grand in the Parisian Quartier Latin, where he excelled in Latin but soon became bored until he developed an interest in mathematics two years later. By the time Galois turned 15, he was devouring the works of famous contemporary mathematicians such as Legendre, Abel, and Lagrange. Not before long he began developing mathematical ideas of his own, neglecting his schoolwork in the process, but leading to his first mathematical paper being published at the age of 17.

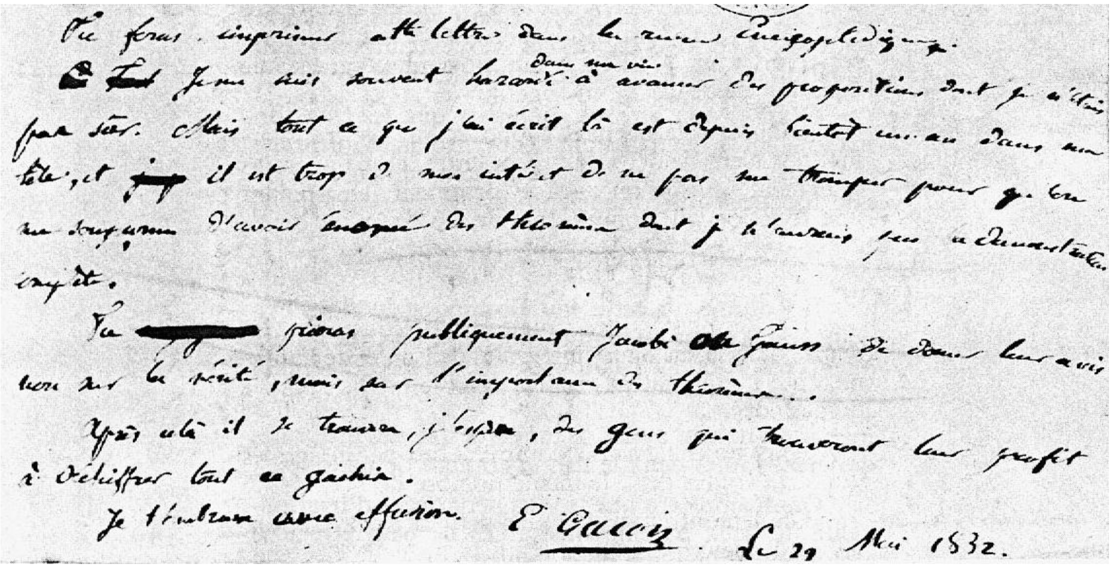
Figure 1. Évariste Galois, age 15, as drawn by a class-mate.

In the spring of 1830 Galois managed to get some more of his work published, but things took a bad turn later that year. At 19 years old, Galois was a typical teenager with strong opinions and hot-headed political ideas. As a fervent republican, Galois clashed with the director of the École Normale to which he had graduated from the Lycée, and got himself expelled from school whereupon he joined the artillery unit of the Republican National Guard. The artillery unit got disbanded by the government and the fresh monarch Louis-Philippe who deemed it too dangerous to have a gang of armed Republicans at large.

Galois' increased political activism severely interfered with his mathematical research, and it was really a stroke of luck when Galois was arrested on July 14 (Bastille Day) 1831 for appearing in public, dressed up in his Republican forbidden artillery unit uniform and armed to the teeth. For during his term of imprisonment until April 29th 1832, he had plenty of time to focus on the further development of his mathematical ideas, which by that time were far beyond the comprehension and imagination of professional mathematicians.

This body of work was never fully developed however. Barely one month after his release from prison, in the morning hours of May 30th 1832, Galois took a shot in the abdomen during a duel at dawn and died the next day at the age of 20. The circumstances of the duel are murky, although there is some modest evidence suggesting that a woman was involved and that it may have been a matter of honor, rivalry, and unrequited infatuation. The story of Galois' untimely demise has been strongly romanticized, aided of course by the lack of reliable historical sources of information. The romantic story would have it that Galois spent the night before the fatal duel frantically scribbling notes about his ideas, in the conviction that he was about to die.

While there is no evidence to support this story, the fact remains that something of the kind must have been preoccupying Galois' mind during the days leading up to May 30th. A seven-page letter dated May 29th to his friend Auguste Chevalier has been preserved (Fig. 2) in which Galois outlines the direction of his many original ideas and vision. In this very letter, Galois observes that one could write several books about those topics, and implores Chevalier to ensure that his mathematical legacy be brought to the attention of those who are able to develop his ideas further.



Tu feras imprimer cette lettre dans la revue encyclopédique.

[illegible] ~~Tout~~ Je me suis souvent hasardé dans ma vie à avancer des propositions dont je n'étais pas sûr. Mais tout ce que j'ai écrit là est depuis bientôt un an dans ma tête, et ~~[illegible]~~ il est trop de mon intérêt de ne pas me tromper pour qu'on me soupçonne d'avoir énoncé des théorèmes dont je n'aurais pas la démonstration complète.

Tu ~~[illegible]~~ prieras publiquement Jacobi ou Gauss de donner leur avis non sur la vérité, mais sur l'importance des théorèmes.

Après cela il se trouvera, j'espère, des gens qui trouveront leur profit à déchiffrer tout ce gâchis.

Je t'embrasse avec effusion. E Galois. Le 29 Mai 1832.

Figure 2. The mathematical testament of Évariste Galois.

It is not until more than a decade after his death however, when in 1846 Galois' work is finally fully printed in the foremost mathematical journal of the day, and he finally gets the recognition he deserves. In the following 1.5 centuries, generations of mathematicians have been exploring Galois' ideas in all their breadth and depth, and the completely new field of mathematics which has so been discovered has been named in his honor: Galois Theory.

Why is this relevant to our quest for a good addition and multiplication on a set of 256 elements? Because the prime area of interest of Galois Theory lies precisely in the description and properties of 'finite fields'. In mathematical terms, a field is a set with an addition and a multiplication. Galois Theory is also referred to as Finite Field Theory. Our set of 256 bytes is a finite set, and we are looking for the way to perform arithmetic on it. In other words, what we are after is the Galois Field of 256 elements!

Given a finite set with q being the number of elements, the Galois Field with q elements is denoted GF(q). The main result from Galois Theory pertaining to our problem is the following: all finite fields are of the form GF(p^n) where p is a prime number and n is a positive integer. More specifically:

- If q = p where p is prime, then GF(p) is a field and the addition and multiplication are the modular operations as discussed in the section on naive approaches above.
- If q = p^n where p is a prime and n is a positive integer, then GF(p^n) is a field and Galois Theory tells us exactly how the addition and multiplication must be defined to make it one. They are not the modular operations however.
- If q is not some power of a prime number, then it is impossible to define an addition and a multiplication to turn the finite set with q elements into a field.

As an example of the second case, consider GF(4), i.e. GF(2^2). The set of all 2-bit bytes (Table 6) has four elements, which is the second power of a prime, two. Galois Theory yields the correct addition and multiplication tables for GF(4):

⊕	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Table 9. The Galois Theory addition on the set of of 2-bit bytes.

⊗	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Table 10. The Galois multiplication on the set of of 2-bit bytes.

Remember that we tried the modular arithmetic on GF(4) and it did not work because 4 is not a prime! Indeed if we compare the above tables to Tables 7 and 8, we observe that we got both of them wrong! As was the case with Coding Theory, it is way out of the scope of this paper to even begin explaining how these tables are constructed with the help of Galois Theory. For the intrepid reader with a rather solid background in mathematics and algebra in particular, we refer to Milne's lecture notes (Milne, 2003) which discuss Galois Theory and many of its applications at length.

One specific aspect of the Galois multiplication tables is worth mentioning though: for the larger finite fields of the form GF(p^n), the ⊗ multiplication is not unique. As an example, GF(2^3) has two possible ⊗ multiplications. That is not a problem though. For the purpose of doing arithmetic on the 8 elements of GF(2^3), one multiplication is as good as the other one, so we can just pick one at whim. The multiplicity of multiplications and the way in which the elements of the multiplication tables are computed, have everything to do with the polynomial representation of the elements as shown in Table 4. But again, not in this paper!

At this point it is time to return to the reason why we were investigating all this: parity blocks on RAID systems are computed by performing arithmetic on the finite set of 2^8 8-bit bytes. That's the eighth power of the prime two, and so Galois Theory gives us the finite field GF(2^8) and tells us what the ⊕ addition and the possible ⊗ multiplications are. Problem solved! In principle, a RAID controller chip needs merely to know the 256 by 256 lookup tables for ⊕ and ⊗ in order to be able to evaluate the expressions from Coding Theory that yield the parity blocks.

It is fairly pointless to fill the remaining pages of this paper with two 255 by 255 tables in a readable font-size, but we can do the following to show the structure of the tables: we assign increasingly dark grey-scale values to each element in the finite field, with white representing the zero-element and black representing the $(p^n\text{-}1)^{th}$ element. Tables 9 and 10 can then be represented as Figures 3 and 4 respectively, and for GF(2^8) the results are shown in Figures 5 and 6.

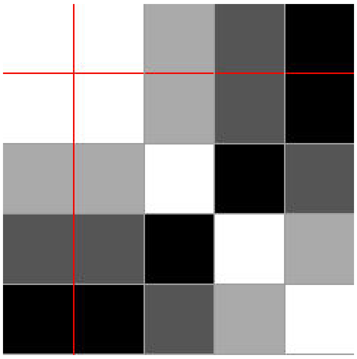


Figure 3. Grey-scale version of ⊕ on GF(4).

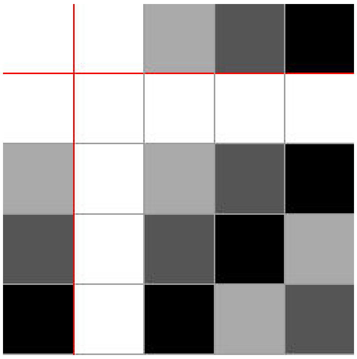


Figure 4. Grey-scale version of ⊗ on GF(4).

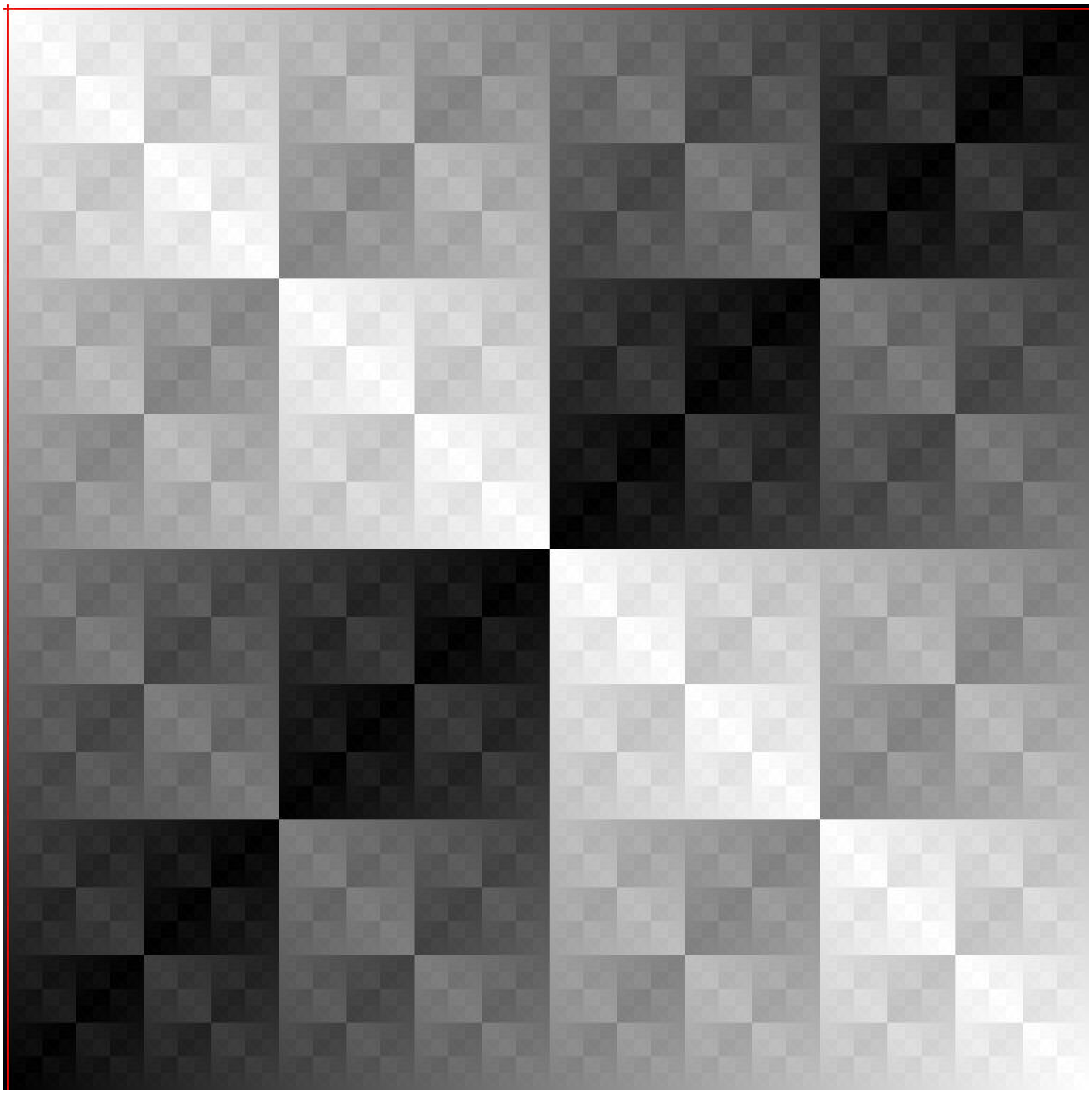


Figure 5. Grey-scale version of ⊕ on GF(256).

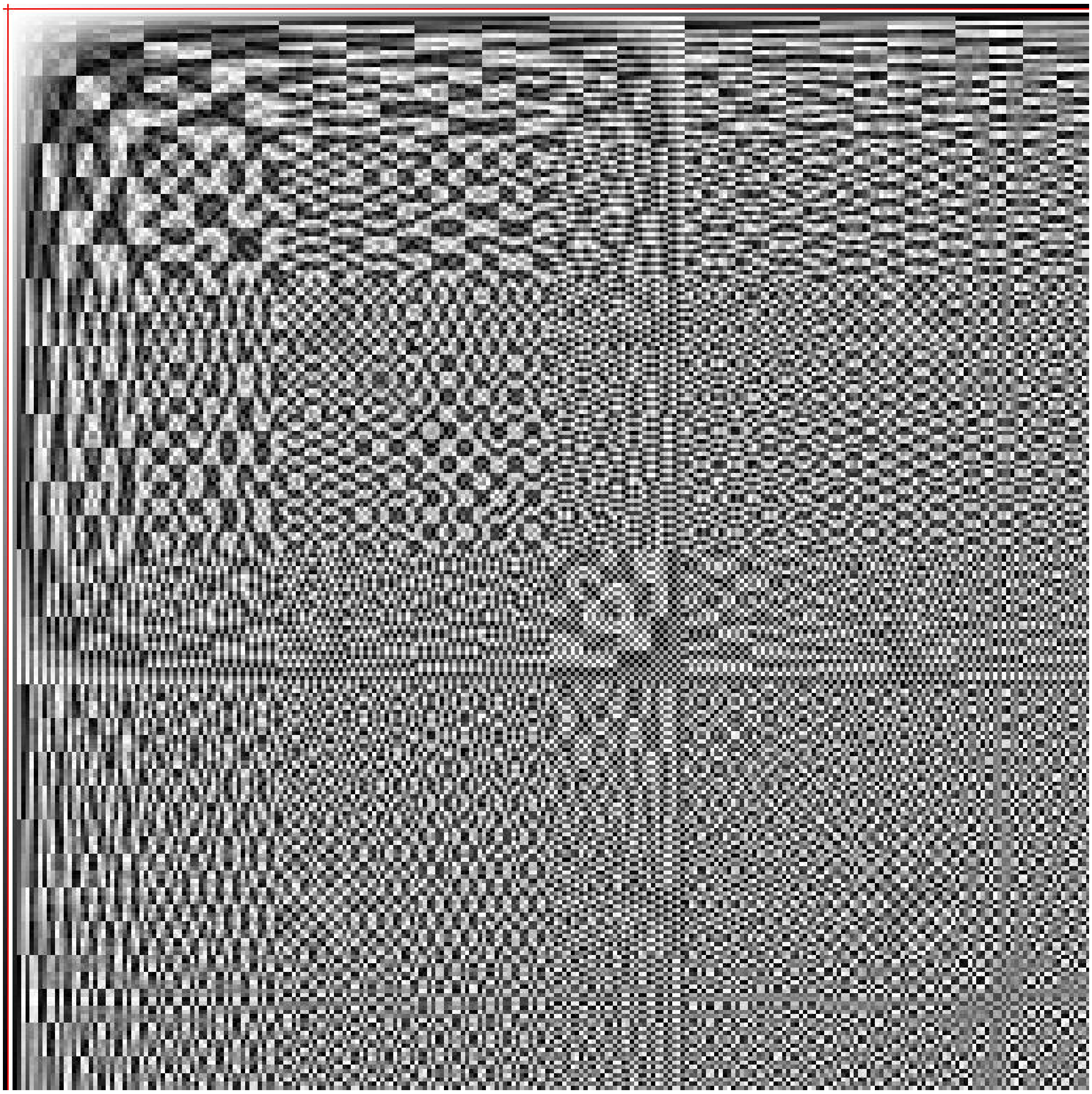


Figure 6. Grey-scale version of ⊗ on GF(256).

Of course, RAID controllers implement ⊕ and ⊗ in a more clever fashion than the performing of look-ups in two 256 by 256 tables. As mentioned above, the expression to calculate the basic RAID parity block as used in RAID-4, RAID-5, and RAID-6, is simply the sum of all the bytes in the data-blocks. E.g. referring back to Table 2, suppose we have a byte B_{32} in the data-block $data_{2,1}$, a byte B_{156} in data-block $data_{2,2}$, byte B_{201} in $data_{2,3}$, and byte B_{45} in $data_{2,4}$. Then the corresponding parity byte in the parity block $parity_2$ is $B_{32} \oplus B_{156} \oplus B_{201} \oplus B_{45}$, which resolves to B_{88} when the table shown in Figure 5 is used.

This sum can be computed much more efficiently though than by performing three table look-ups on two search-keys. It can be shown that the bitwise exclusive OR operator (BXOR) yields the exact same result. There are deep reasons why this is the case, which we shall not expand upon here. The BXOR operator definition is shown in Table 11.

BXOR	0	1
0	0	1
1	1	0

Table 11. The bitwise exclusive OR operator.

If we apply this to the set of 2-bit bytes as listed in Table 6, we get the following:

BXOR	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

Table 12. Bitwise exclusive OR applied to the set of of 2-bit bytes.

We see that BXOR yields the exact same look-up table as the ⊕ look-up table derived from Galois Theory (Table 9). Therefore, whenever bytes need to be added up, it suffices to BXOR the lot. And that is really advantageous because BXOR is a bit-level operation CPU's can do extremely fast.

When it comes to implementations of the ⊗ multiplication, RAID performance is dramatically improved by using two one-dimensional look-up tables of size 256 containing logarithms and inverse logarithms. Indeed, also on a Galois Field, multiplication of two elements can be done by taking the inverse logarithm of the sum of the logarithms of both elements. So two single-key look-up arrays of 256 elements are used, rather than 1 dual-key look-up table of size 2^{16}, which boosts efficiency. The construction of these lists of logarithms and inverse logarithms is documented by Plank (1997).

A BASE SAS RAID CONTROLLER SIMULATION

To illustrate the main ingredients of the story presented so far, we construct a RAID algorithm in Base SAS code. Since RAID-5 is the most ubiquitous it might be nice to show how RAID-5 can be done. However, we decided that adding the rotating scheme to deposit the parity blocks throughout the disk array had an obfuscating effect on the readability of the SAS code. Since we aim to show how it works, we opted for RAID-4 where all parity blocks are stored on the same disk in the array.

PRELIMINARIES

First we need to take care of some basic set-up, and generate a file which we'll use to write to our RAID-4 disk array. We set the root directory for the application, and define four global macro variables (which we conventionally prefix with g_) pointing to four directories which will act as 'disks' in a 4-disk RAID-4 array.

```
%let g_rootpath=c:\temp\french connection;
%let g_output=&g_rootpath.\output;
%let g_disk1=&g_rootpath.\data\volumes\raid5\disk1;
%let g_disk2=&g_rootpath.\data\volumes\raid5\disk2;
%let g_disk3=&g_rootpath.\data\volumes\raid5\disk3;
%let g_disk4=&g_rootpath.\data\volumes\raid5\disk4;
```

Then a library to store some output later:

```
libname output "&g_output";
```

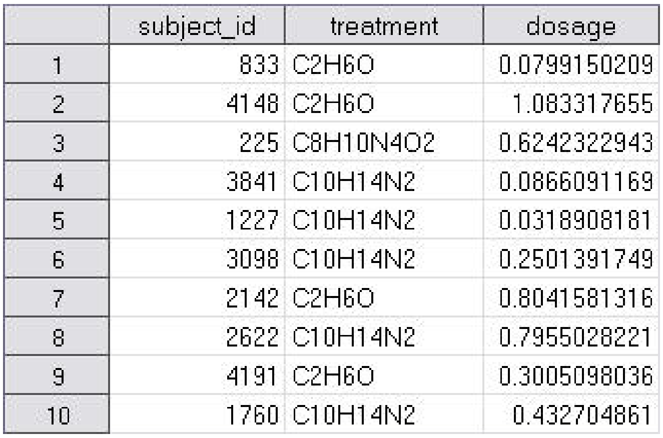
We create a data set WORK.SAMPLE in clinical trials fashion. First choose the desired number of records, and the desired average sparseness of the key, then generate a bunch of Subject ID's, give each of them a certain dosage of three possible treatments:

```
%let g_max_subjects=900;
%let g_avg_sparseness=5;
%let g_factor=%eval(&g_max_subjects*&g_avg_sparseness);

data sample(drop=i rnd);
  length
    i
    dosage
    rnd
    subject_id    8
    treatment  $ 10
    ;
  do i=1 to &g_max_subjects;
    subject_id=ceil(ranuni(1)*&g_factor);
    rnd=ranpoi(1,10);
    if rnd le 5 then treatment='C8H10N4O2';
    else if rnd gt 12 then treatment='C2H6O';
    else treatment='C10H14N2';
    dosage=abs(rannor(10));
    output;
    end;
run;
```

And we define a fileref SOMEFILE pointing to our test-file.

```
data _null_;
  length workpath $ 200;
  workpath=pathname('work');
  call symput('g_workpath',trim(left(workpath)));
run;

filename somefile "&g_workpath\sample.sas7bdat";
```

The first few records of WORK.SAMPLE are shown in Figure 7.



	subject_id	treatment	dosage
1	833	C2H6O	0.0799150209
2	4148	C2H6O	1.083317655
3	225	C8H10N4O2	0.6242322943
4	3841	C10H14N2	0.0866091169
5	1227	C10H14N2	0.0318908181
6	3098	C10H14N2	0.2501391749
7	2142	C2H6O	0.8041581316
8	2622	C10H14N2	0.7955028221
9	4191	C2H6O	0.3005098036
10	1760	C10H14N2	0.432704861

Figure 7. The first 10 observations in WORK.SAMPLE.

Inspecting the file-system properties of SAMPLE.SAS7BDAT, we see that the file-size is 33,792 bytes, which is exactly 33 times 1024 bytes.

RAID-4 WRITE

We then proceed to write our sample file to our RAID-4 array. In summary, the idea is this: since we have three disks to store data-blocks, we read our file as three blocks of 1024 bytes per iteration of the DATA step. These will be our RAID data-blocks. To compute the parity block, we convert these strings of 1024 bytes to strings of 8192 bits. The bit-strings are then loaded into arrays, which facilitates explicitly coding the BXOR logic, the result of which is a new array containing the parity bits. These are strung together into a bit-string of length 8192, which is converted to 1024 bytes forming the parity block. The data-blocks are written to Disks 1 through 3, and the parity block goes to Disk 4. The iteration number _N_ of the DATA step is used to number and identify the stripes. The main programming challenge here is to get all that done in a single DATA step:

```
data _null_;
  put _n_=;
  length bytes1 bytes2 bytes3 rbytes $ 1024;
  length bits1 bits2 bits3 rbits $ 8192;
  length addr_arr_bits1 addr_arr_bits2 addr_arr_bits3 addr_arr_rbits arr_idx 8;
  length fname1 fname2 fname3 fname4 $ 255;
  array arr_bits1[1:8192] $ 1;
  array arr_bits2[1:8192] $ 1;
  array arr_bits3[1:8192] $ 1;
  array arr_rbits[1:8192] $ 1;
  infile somefile recfm=f lrecl=1024;❶
  input bytes1 1-1024;❷
  input bytes2 1-1024;
```

```
    input bytes3 1-1024;
    bits1=put(bytes1,$binary8192.);❸
    bits2=put(bytes2,$binary8192.);
    bits3=put(bytes3,$binary8192.);
    addr_arr_bits1=addr(arr_bits1[1]);
    addr_arr_bits2=addr(arr_bits2[1]);
    addr_arr_bits3=addr(arr_bits3[1]);
    addr_arr_rbits=addr(arr_rbits[1]);
    call poke(bits1,addr_arr_bits1,8192);❹
    call poke(bits2,addr_arr_bits2,8192);
    call poke(bits3,addr_arr_bits3,8192);
    do arr_idx=1 to 8192;
      if❺
       (arr_bits1[arr_idx]='1' and arr_bits2[arr_idx]='0' and arr_bits3[arr_idx]='0')
       or
       (arr_bits1[arr_idx]='0' and arr_bits2[arr_idx]='1' and arr_bits3[arr_idx]='0')
       or
       (arr_bits1[arr_idx]='0' and arr_bits2[arr_idx]='0' and arr_bits3[arr_idx]='1')
       or
       (arr_bits1[arr_idx]='1' and arr_bits2[arr_idx]='1' and arr_bits3[arr_idx]='1')
      then arr_rbits[arr_idx]='1';
      else arr_rbits[arr_idx]='0';
      end;
    rbits=peekc(addr_arr_rbits,8192);❻
    rbytes=input(rbits,$binary8192.);❼
    fname1="&g_disk1\stripe"||trim(left(put(_n_,z3.)))||".bin";❽
    fname2="&g_disk2\stripe"||trim(left(put(_n_,z3.)))||".bin";
    fname3="&g_disk3\stripe"||trim(left(put(_n_,z3.)))||".bin";
    fname4="&g_disk4\stripe"||trim(left(put(_n_,z3.)))||".bin";
    file disk1 filevar=fname1 lrecl=1024 recfm=n;❾
    put bytes1;
    file disk2 filevar=fname2 lrecl=1024 recfm=n;
    put bytes2;
    file disk3 filevar=fname3 lrecl=1024 recfm=n;
    put bytes3;
    file disk4 filevar=fname4 lrecl=1024 recfm=n;❿
    put rbytes;
  run;
```

A few noteworthy details about the code:

❶ We read the file with RECFM=F and LRECL=1024 to ensure that we get fixed blocks of 1024 bytes each.

❷ Three consecutive blocks of 1024 bytes are read and stored in the character variables BYTES1 through BYTES3. Column input is used, specifying again that the full content of the _INFILE_ buffer must be placed in the variable, lest the data should contain a delimiter leading to bytes being dropped.

❸ Bytes are converted into strings of bits.

❹ The bit-strings are loaded into arrays consisting of single-byte character variables. So each array-element contains a character ‘0’ or ‘1’ value. We use CALL POKE to achieve this, since it is the most efficient way to do this and avoids coding DO loops over an array index. We refer those unfamiliar with the technique to Dorfman (2009) where several safe and practical uses of peeking and poking are revealed, including this one.

❺ The condition on the IF statement reflects the four cases where applying BXOR to three bits evaluates to 1. In general, BXOR over a number of bits returns 1 if an odd number of bits are set to 1. In our case: either all three array-elements contain ‘1’, or only one of them contains ‘1’. Note that SAS has its own BXOR function. For educational purposes we decided not to use it here and code the BXOR logic explicitly.

❻ PEEKC is used to string all the redundancy bits now present in the ARR_RBITS array together into a character string in a simple one-liner. Again, we refer to Dorfman (2009) for the details.

❼ The redundancy bits are transformed into a block of 1024 bytes.

❽ Character variables FNAME1 through FNAME4 are constructed to contain the output path and file-name for the current RAID-stripe.

❾ By using the FILEVAR option we ensure that the three data-blocks for the current stripe are written to the correct Disk number.

❿ And in similar vein, the newly computed parity block is written to Disk 4.

Last but not least, it should be noted that this code works fine with the provided test-file because its file-size is 33 times 1024 bytes, meaning that the entire file is being read in 11 iterations of the DATA step with all three INPUT variables BYTES1 through BYTES3 being filled with data. The code will not work for files where the file-size is not a multiple of 3096 bytes. In those cases, more logic must be added to accommodate the fact that either BYTES2 or BYTES3, or both, might not contain data...

DISASTER STRIKES!

We simulate a hard disk crash in which Disk 2 expires. We look for the Disk 2 directory on the file-system, and delete all of its contents. One third of the bytes making up our test-file is now effectively missing!

RESURRECTION

We rebuild the contents of Disk 2 by using the remaining data-blocks on Disk 1 and Disk 3, and the parity blocks on Disk 4.

```
filename dir "&g_disk1";❶

data contents_disk1(keep=filename);
  length filename $ 200;
  did=dopen('dir');
  do i=1 to dnum(did);
    filename=dread(did,i);
    output;
    end;
  rc=dclose(did);
run;

data _null_;
  set contents_disk1 (obs=1) nobs=nobs;
  call sympput('g_n_stripes',trim(left(put(nobs,8.))));❷
run;

data _null_;
  length bytes1 bytes2 bytes3 rbytes $ 1024;
  length bits1 bits2 bits3 rbits $ 8192;
  length addr_arr_bits1 addr_arr_bits2 addr_arr_bits3 addr_arr_rbits 8;
  length arr_idx stripe_num 8;
  length fname1 fname2 fname3 fname4 $ 255;
  array arr_bits1[1:8192] $ 1;
  array arr_bits2[1:8192] $ 1;
  array arr_bits3[1:8192] $ 1;
  array arr_rbits[1:8192] $ 1;
  do stripe_num=1 to &g_n_stripes;❸
    fname1="&g_disk1\stripe"||trim(left(put(stripe_num,z3.)))||".bin";
    fname2="&g_disk2\stripe"||trim(left(put(stripe_num,z3.)))||".bin";
    fname3="&g_disk3\stripe"||trim(left(put(stripe_num,z3.)))||".bin";
    fname4="&g_disk4\stripe"||trim(left(put(stripe_num,z3.)))||".bin";
    infile disk1 filevar=fname1 lrecl=1024 recfm=f;
    input bytes1 1-1024;
    infile disk3 filevar=fname3 lrecl=1024 recfm=f;
    input bytes3 1-1024;
    infile disk4 filevar=fname4 lrecl=1024 recfm=f;
    input rbytes 1-1024;
    bits1=put(bytes1,$binary8192.);
    bits3=put(bytes3,$binary8192.);
    rbits=put(rbytes,$binary8192.);
    addr_arr_bits1=addr(arr_bits1[1]);
    addr_arr_bits2=addr(arr_bits2[1]);
    addr_arr_bits3=addr(arr_bits3[1]);
    addr_arr_rbits=addr(arr_rbits[1]);
    call poke(bits1,addr_arr_bits1,8192);
    call poke(bits3,addr_arr_bits3,8192);
    call poke(rbits,addr_arr_rbits,8192);
    do arr_idx=1 to 8192;
     if
     (arr_bits1[arr_idx]='1' and arr_bits3[arr_idx]='0' and arr_rbits[arr_idx]='0')
     or
     (arr_bits1[arr_idx]='0' and arr_bits3[arr_idx]='1' and arr_rbits[arr_idx]='0')
     or
     (arr_bits1[arr_idx]='0' and arr_bits3[arr_idx]='0' and arr_rbits[arr_idx]='1')
     or
     (arr_bits1[arr_idx]='1' and arr_bits3[arr_idx]='1' and arr_rbits[arr_idx]='1')
     then arr_bits2[arr_idx]='1';
     else arr_bits2[arr_idx]='0';
     end;
    bits2=peekc(addr_arr_bits2,8192);
    bytes2=input(bits2,$binary8192.);
    file disk2 filevar=fname2 lrecl=1024 recfm=n;
    put bytes2;
    end;
  stop;
run;
```

❶ We will need a DO loop over the number of stripes in order to reconstruct the missing data-block for each stripe, so we look at the contents of Disk 1, which was not affected by the disaster.

❷ The number of stripes found by inspecting Disk 1 is stored into a global macro variable G_N_STRIPES.

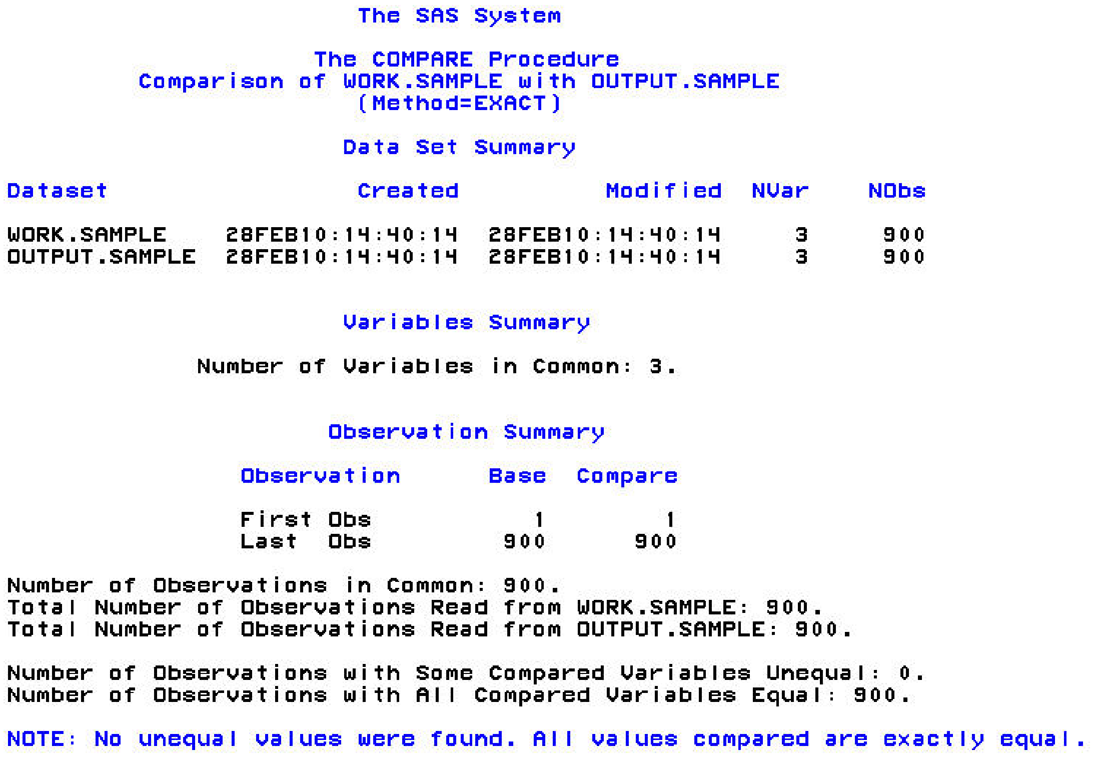
❸ The DO loop runs for each stripe, and utilizes the same logic and constructs as detailed in the RAID-4 Write code to BXOR the remaining data-blocks BYTES1 and BYTES3, as well as the parity block RBYTES, and so reconstruct the missing data-block BYTES2 and write it to the Disk 2 location.

RAID-4 READ

Finally we reassemble our sample file by recombining the original data-blocks on Disk 1 and Disk 3, and the resurrected data-blocks on Disk 2. We show that the resulting file is identical to the data set we started from.

```
filename outfile "&g_output\sample.sas7bdat" mod;

data _null_;
  length bytes1 bytes2 bytes3 rbytes $ 1024;
  length stripe_num 8;
  length fname1 fname2 fname3 $ 255;
  file outfile lrecl=1024 recfm=n;
  do stripe_num=1 to &g_n_stripes;
    fname1="&g_disk1\stripe"||trim(left(put(stripe_num,z3.)))||".bin";
    fname2="&g_disk2\stripe"||trim(left(put(stripe_num,z3.)))||".bin";
    fname3="&g_disk3\stripe"||trim(left(put(stripe_num,z3.)))||".bin";
    infile disk1 filevar=fname1 lrecl=1024 recfm=n;
    input bytes1 1-1024;
    infile disk2 filevar=fname2 lrecl=1024 recfm=n;
    input bytes2 1-1024;
    infile disk3 filevar=fname3 lrecl=1024 recfm=n;
    input bytes3 1-1024;
    put bytes1;
    put bytes2;
    put bytes3;
    end;
  stop;
run;

proc compare base=work.sample comp=output.sample;
run;
```

As Figure 8 demonstrates, the original data set WORK.SAMPLE and the reconstructed OUTPUT.SAMPLE are exactly equal. The COMPARE procedure reports the ‘Created’ and ‘Modified’ dates as identical because it reads this information from the respective data set headers, and not from the file-system where the OUTPUT.SAMPLE data set is of course reported as being more recent than the WORK.SAMPLE one...



```
                        The SAS System

                     The COMPARE Procedure
         Comparison of WORK.SAMPLE with OUTPUT.SAMPLE
                        (Method=EXACT)

                       Data Set Summary

Dataset                 Created          Modified  NVar    NObs

WORK.SAMPLE    28FEB10:14:40:14  28FEB10:14:40:14     3     900
OUTPUT.SAMPLE  28FEB10:14:40:14  28FEB10:14:40:14     3     900

                       Variables Summary

             Number of Variables in Common: 3.


                      Observation Summary

                Observation      Base  Compare

                First Obs           1        1
                Last  Obs         900      900

Number of Observations in Common: 900.
Total Number of Observations Read from WORK.SAMPLE: 900.
Total Number of Observations Read from OUTPUT.SAMPLE: 900.

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 900.

NOTE: No unequal values were found. All values compared are exactly equal.
```

Figure 8. Output from PROC COMPARE.

CONCLUSION

We have demonstrated how RAID-4 works in SAS code, and more importantly, we have shown that some seriously strange yet beautiful mathematics first conceived two centuries ago in the mind of a brilliant young man is what really makes today’s RAID storage systems tick! The reader eager to experiment further might try to simulate RAID-5 by tweaking the SAS code to rotate the output file locations over the available disks in the array. Some more detail will then be needed in the naming scheme for the parts of stripes, since one would need to know which parts are data-blocks, and which parts are parity blocks. For the rotational aspect and the construction of output file names, the MOD function may be used. And what about RAID-6? Implementing RAID-6 in Base SAS code would be vastly more complex, and might perhaps be food for a follow-up paper. As already mentioned, RAID controller chips typically use LOG and LOG^{-1} look-up tables instead of multiplication and division look-up tables. A potentially fruitful strategy might be to code these look-ups by means of hash objects in the DATA step.

REFERENCES

Anvin H.P., “The Mathematics of RAID6”, *online paper*, 2009.
Available at: http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf

Dorfman P.M., “From Obscurity to Utility: ADDR, PEEK, POKE as DATA Step Programming Tools”, *Proceedings of the SAS Global Forum 2009 Conference*, Washington D.C., paper 010, 2009.
Available at: http://support.sas.com/resources/papers/proceedings09/010-2009.pdf

Milne J.S., “Fields and Galois Theory”, *online lecture notes*, 2003.
Available at: http://www.galois-group.net/theory/math594fS.pdf

Patterson D.A., Gibson G., Katz R.H., “A Case for Redundant Arrays of Inexpensive Disks (RAID)”, *Proc. ACM SIGMOD Conf.*, Chicago, 1988, p. 109.
Available at: http://www.cs.cmu.edu/~garth/RAIDpaper/Patterson88.pdf

Plank J.S., “A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems”, *Software—Practice and Experience (SPE)*, 27(9), 1997, pp. 955-1012.
Available at: http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.pdf

RECOMMENDED READING

- The Galois Translation Project
  http://www.galois-group.net/gtp/index//EN/

  For those so inclined, we recommend having a look at the various surviving historical documents including letters by and about Galois, which are available through the Galois Translation Project site. Some of these have indeed been translated already, others are still only available in French. They provide fascinating reading, although things become a touch macabre when one reads the coroner's report of the post-mortem on Galois' body and the detailed description of his wounds and extracted brains...

- "Finite Field Tables" from The Wolfram Demonstrations Project
  http://demonstrations.wolfram.com/FiniteFieldTables/

  The Wolfram Demonstrations Project offers an ever-expanding collection of fascinating Mathematica based applications, including quite a few on the topic of Galois Fields. We recommend experimenting with the wonderful "Finite Field Tables" demonstration as contributed by Ed Pegg Jr. We used Mathematica 7 and modified source code from the "Finite Field Tables" demonstration to produce Figs. 5 and 6 for the field GF(2^8).

ACKNOWLEDGEMENTS

The authors should like to thank Karst Gelissen of the SAS Netherlands Technical Support team for his enlightening lectures on encodings and other esoteric topics pertaining to the handling of binary data within the DATA step, even though these were diligently avoided mentioning in the present paper. Special gratitude is also due to Tanya Kalich of the SAS, Inc. Technical Support team for careful proofreading, valuable suggestions, and above all: interesting discussions about Galois Theory!

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Koen Vyverman
SAS Institute B.V.
Flevolaan 69
1272PC Huizen
The Netherlands
Email: support@snl.sas.com

Paul M. Dorfman
4437 Summer Walk Ct.,
Jacksonville, FL 32258
(904) 260-6509
Email: sashole@bellsouth.net