Tutorial on Erasure Coding for Storage Applications, Part 1

James S. Plank

Professor
EECS Department
University of Tennessee
plank@cs.utk.edu

Usenix FAST
February 12, 2013

Historical Perspective

Coding Theory — Reed Solomon Codes, Error Correction
Colossal, math-laden field

Storage — RAID-6: EVENODD, RDP, X-Code
Specialized XOR-based codes

Graph Theory — LDPC Codes (Tornado, Raptor, LT)
XOR codes for content distribution

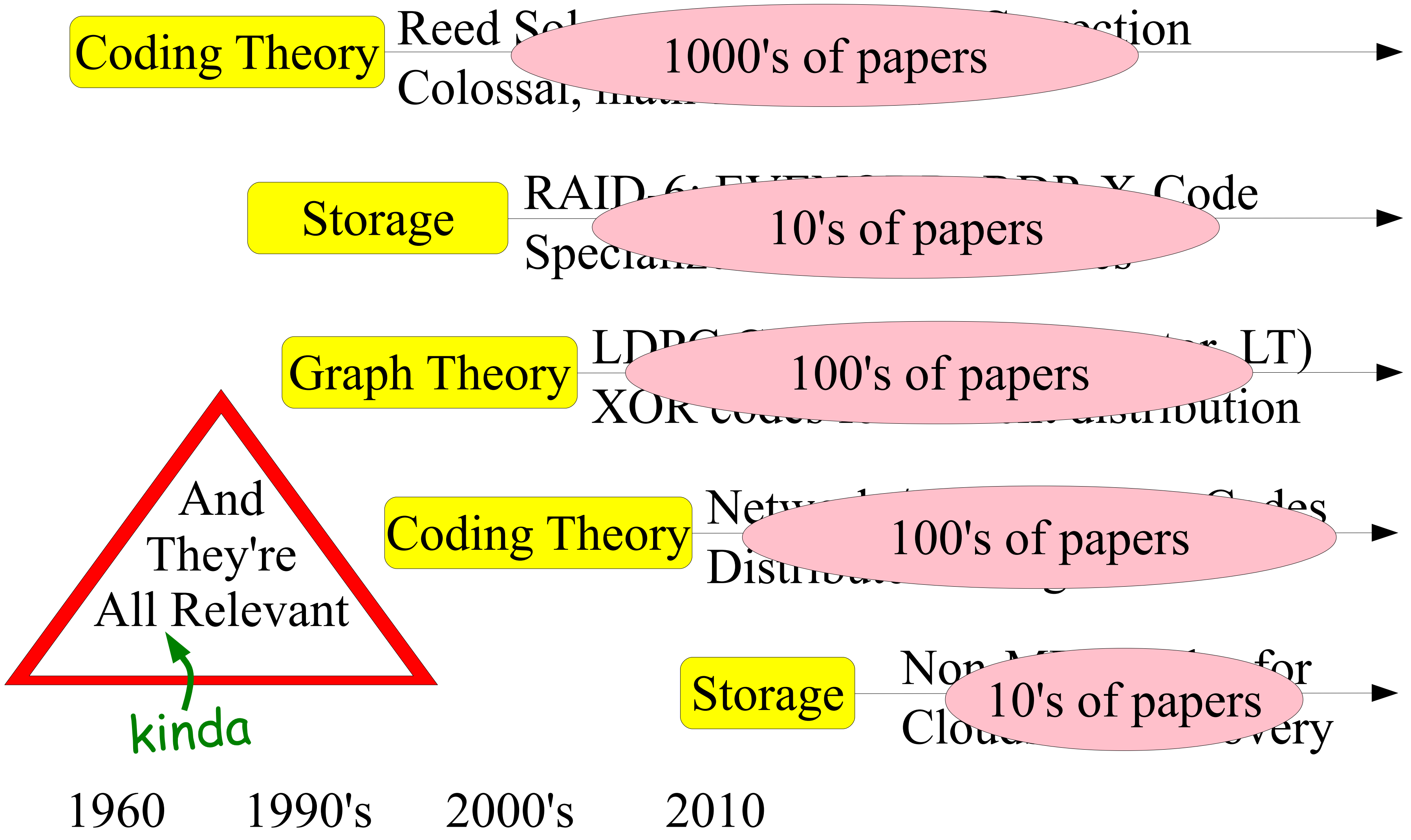
Coding Theory — Network / Regenerating Codes
Distributed storage

Storage — Non-MDS codes for
Clouds and Recovery

You Are Here

1960 1990's 2000's 2010

Historical Perspective



Our Mission in this Tutorial, Part 1

- Give you the basics of coding for storage
  - Nomenclature, techniques
  - Matrices, Finite Field Arithmetic
- Present the classics
  - Reed-Solomon Codes
  - RAID-6 Classics: EVENODD/RDP/X-Code
  - Generalized XOR Codes
- Inform you of open source solutions

Our Mission in this Tutorial, Part 2

- Inform you of more recent, non-classic codes
  - Regenerating Codes
  - Techniques for improved recovery
  - Non-MDS codes for cloud storage
  - Erasure coding for Flash

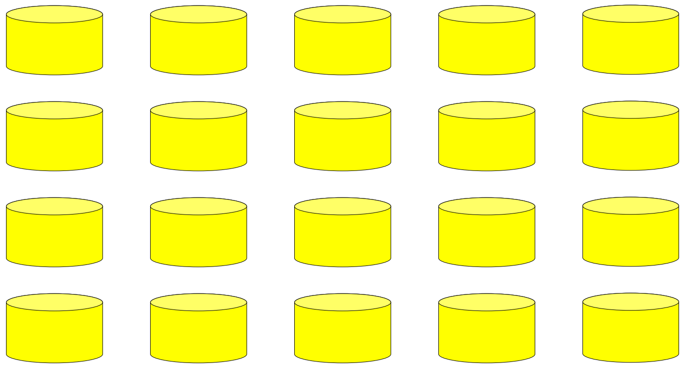
My Philosophy for Part 1

- Hold your hand in the beginning
- Then swamp you with information
- Make sure you have reference material.

Let's Dig in: Erasure Coding Basics

You've got some data:

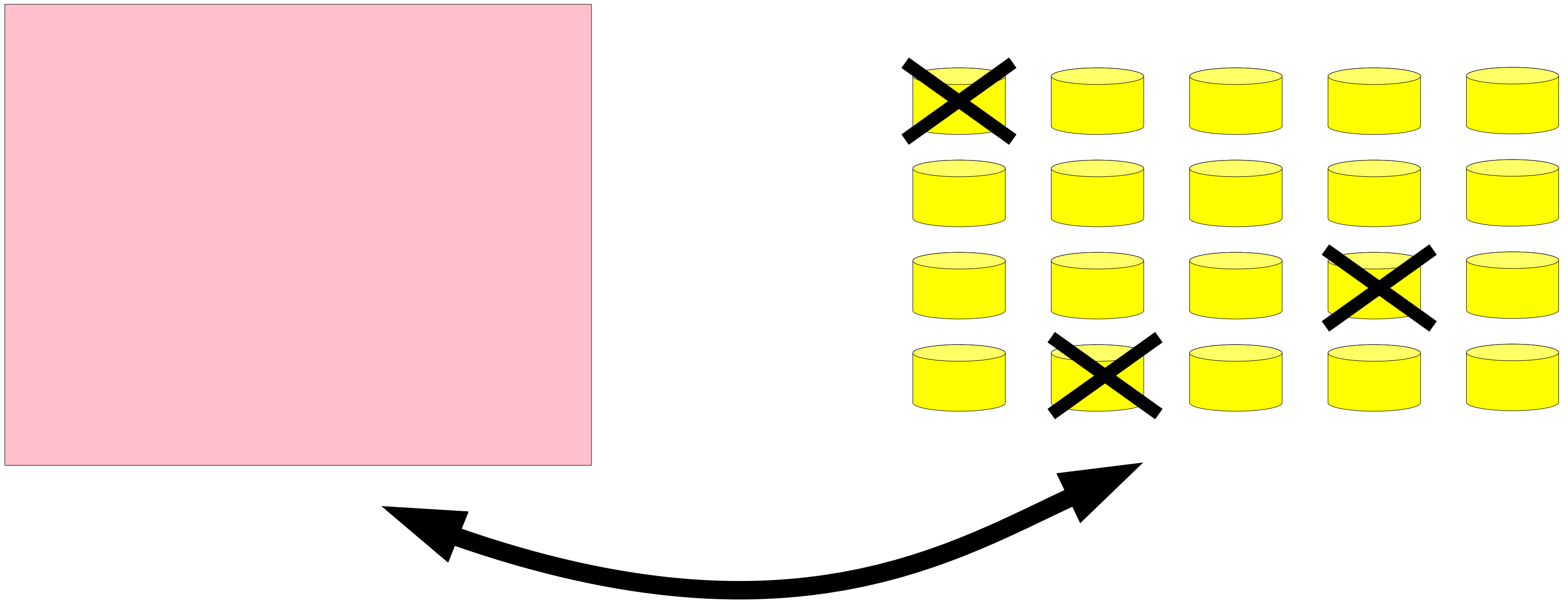
And a collection of storage nodes.



Let's Dig in: Erasure Coding Basics

You've got some data:

And a collection of storage nodes.

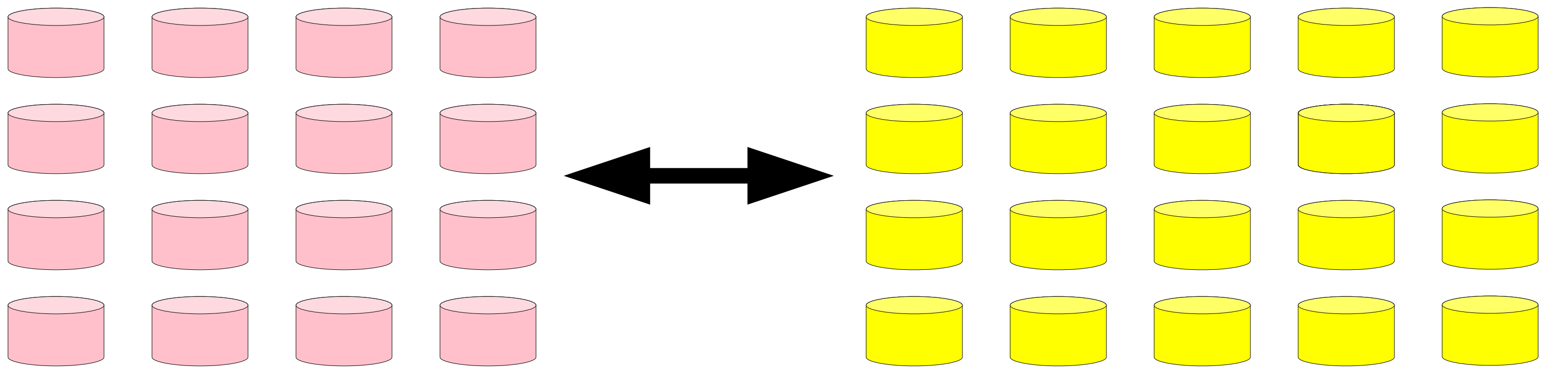


And you want to store the data on the storage nodes so that you can get the data back, even when the nodes fail.

Let's Dig in: Erasure Coding Basics

More concrete: You have *k* disks worth of data:

And *n* total disks.

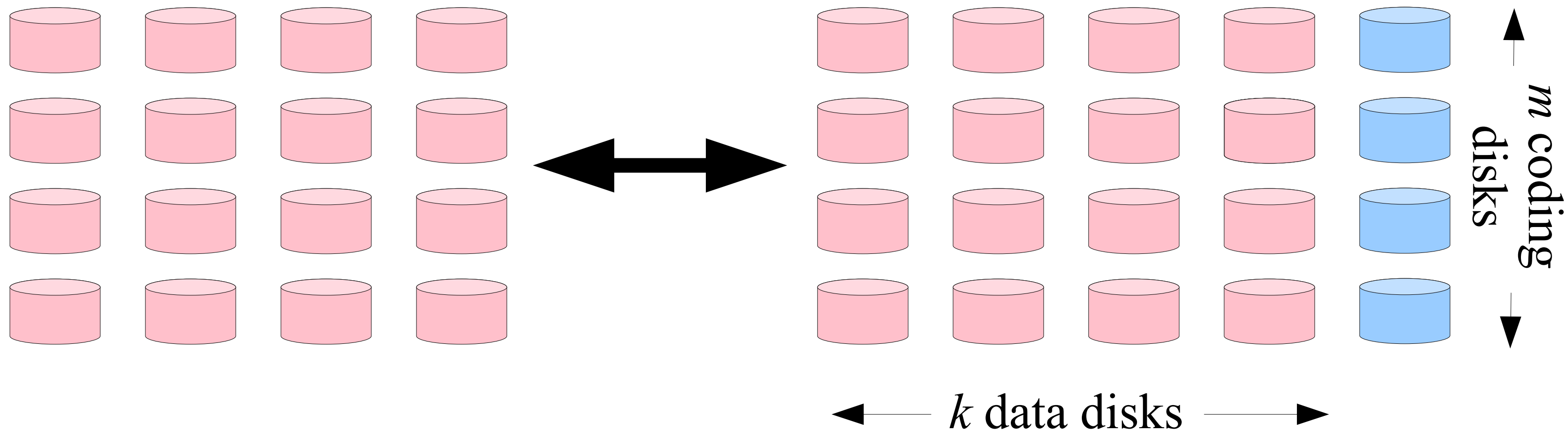


The erasure code tells you how to create *n* disks worth of data+coding so that when disks fail, you can still get the data.

Nomenclature

You have
k disks worth of data:

And n total disks.
$n = k + m$

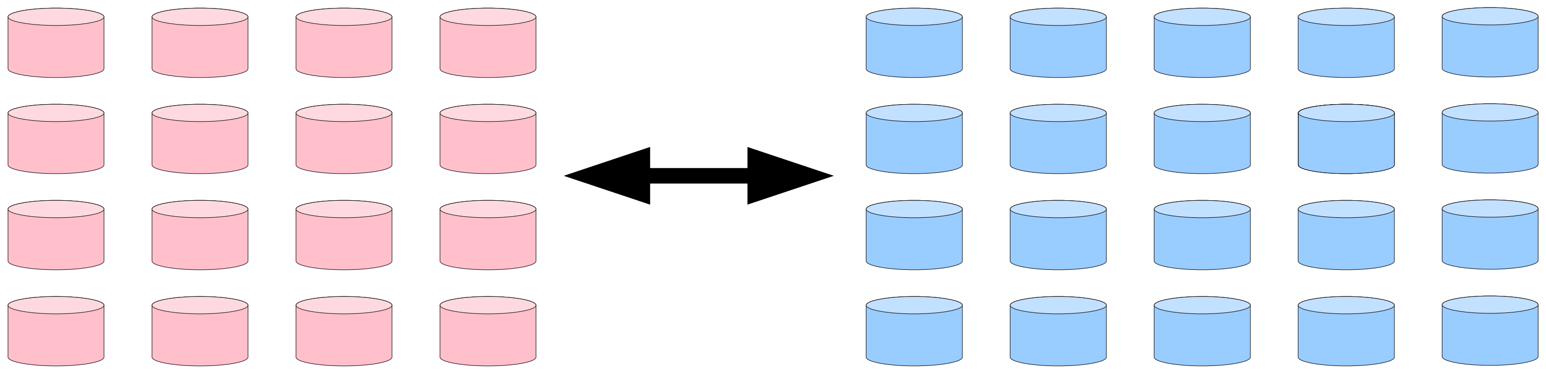


A systematic erasure code stores the data in the clear on k of the n disks.
There are k data disks, and m coding or “parity” disks.

Nomenclature

You have
k disks worth of data:

And *n* total disks.
$n = k + m$

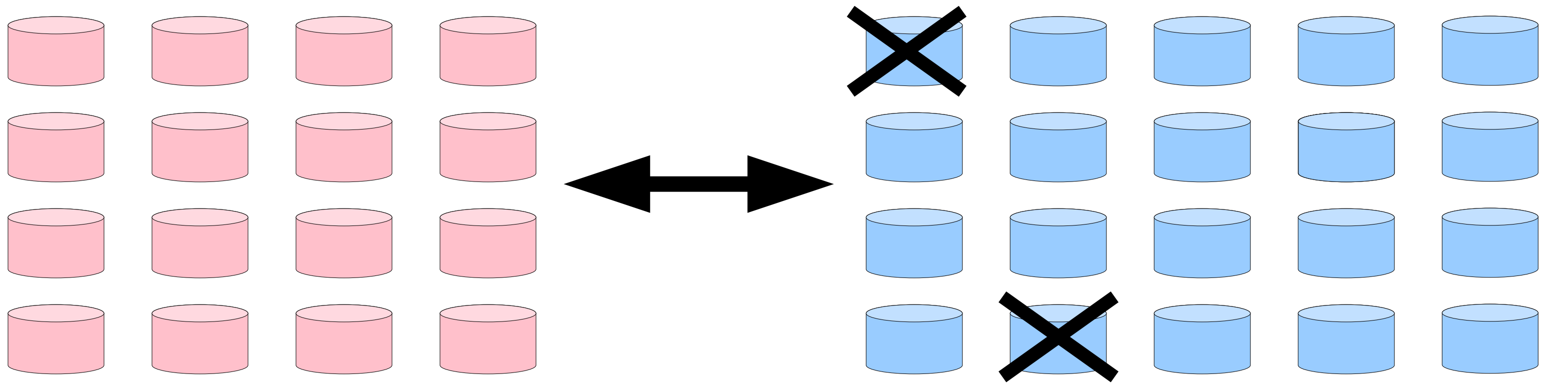


A non-systematic erasure code stores only coding information, but we still use *k*, *m*, and *n* to describe the code.

Nomenclature

You have
k disks worth of data:

And *n* total disks.
$n = k + m$

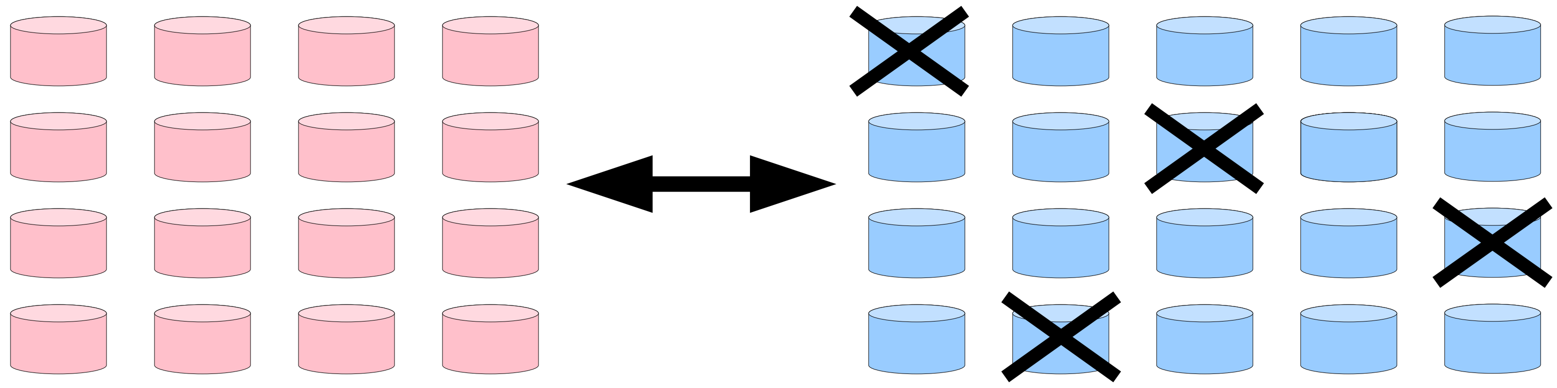


When disks fail, their contents become unusable,
and the storage system detects this.
This failure mode is called an erasure.

Nomenclature

You have
k disks worth of data:

And n total disks.
$n = k + m$



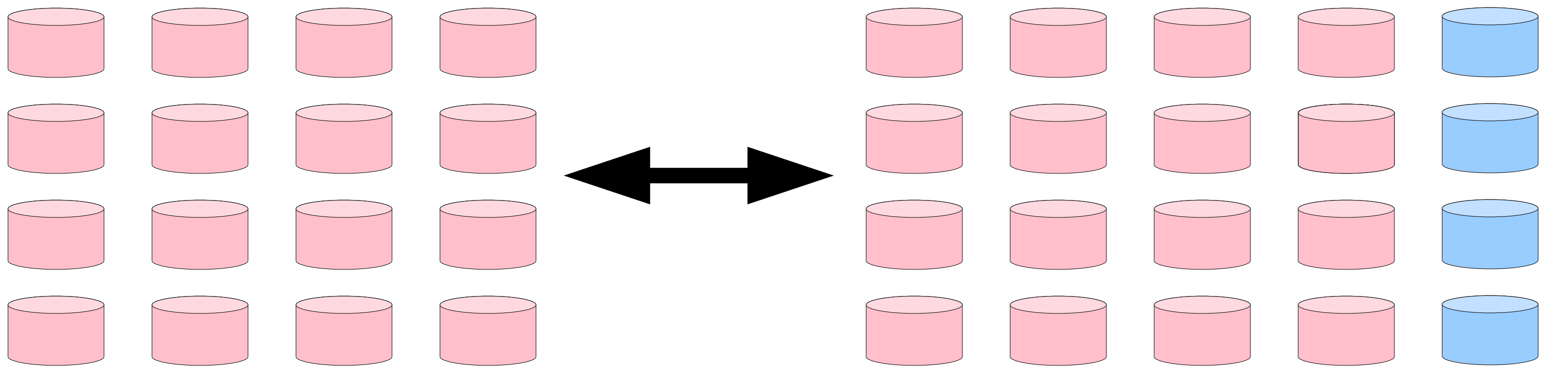
An MDS ("Maximum Distance Separable") code can reconstruct the data from any m failures.

That's as good as you can do.

Nomenclature

You have
k disks worth of data:

And n total disks.
$n = k + m$

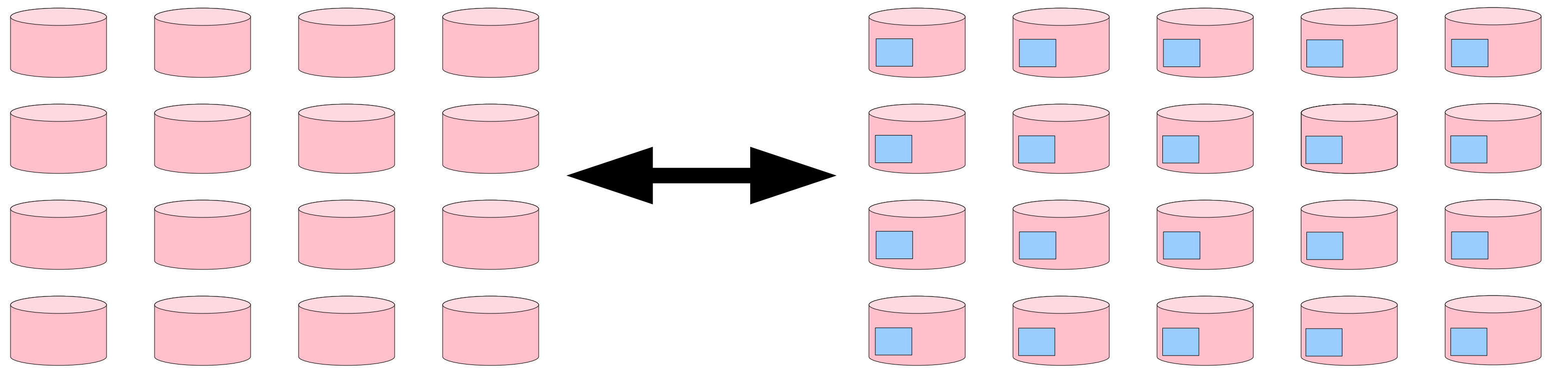


A horizontal code is systematic,
and partitions the disks into data disks and coding disks.

Nomenclature

You have
k disks worth of data:

And *n* total disks.
$n = k + m$

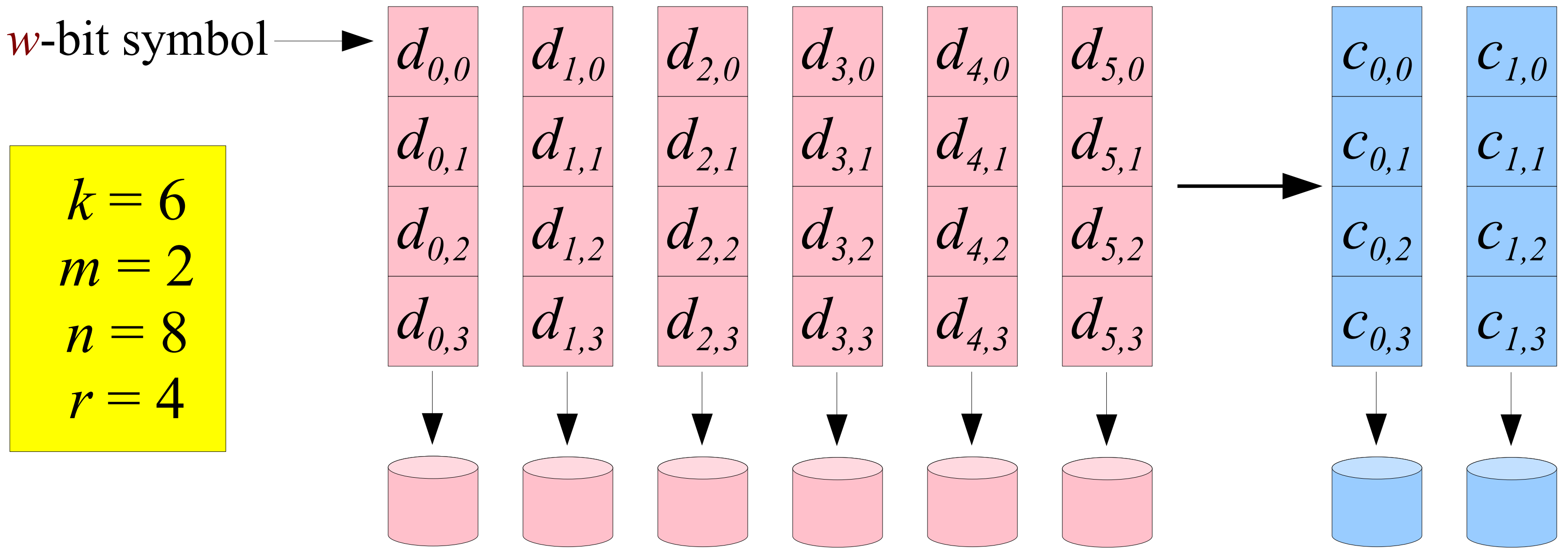


A vertical code is systematic, but
each disk is required to hold some data and some coding.

Vertical codes can be MDS, so long
as they tolerate all combinations of *m* failed disks.

Two Views of a "Stripe"

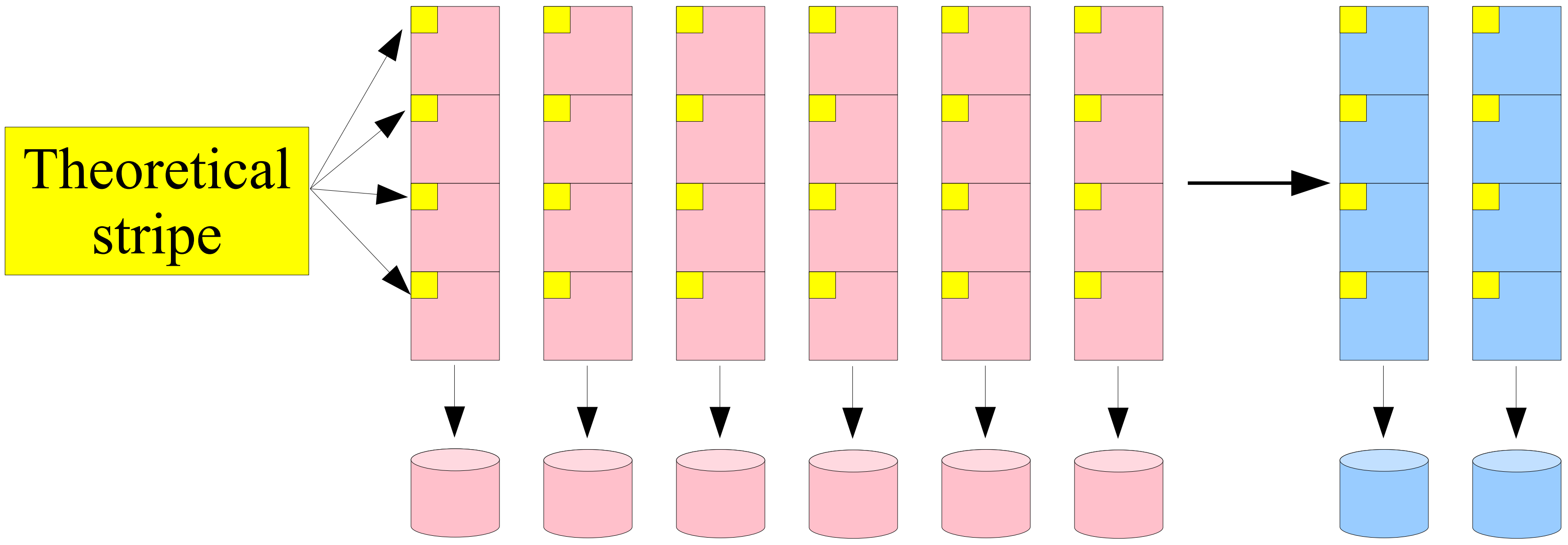
- The Theoretical View:
  - The minimum collection of bits that encode and decode together.
  - *r* rows of *w*-bit symbols from each of *n* disks:



w-bit symbol
$k = 6$
$m = 2$
$n = 8$
$r = 4$
$d_{0,0}$ $d_{0,1}$ $d_{0,2}$ $d_{0,3}$
$d_{1,0}$ $d_{1,1}$ $d_{1,2}$ $d_{1,3}$
$d_{2,0}$ $d_{2,1}$ $d_{2,2}$ $d_{2,3}$
$d_{3,0}$ $d_{3,1}$ $d_{3,2}$ $d_{3,3}$
$d_{4,0}$ $d_{4,1}$ $d_{4,2}$ $d_{4,3}$
$d_{5,0}$ $d_{5,1}$ $d_{5,2}$ $d_{5,3}$
$c_{0,0}$ $c_{0,1}$ $c_{0,2}$ $c_{0,3}$
$c_{1,0}$ $c_{1,1}$ $c_{1,2}$ $c_{1,3}$

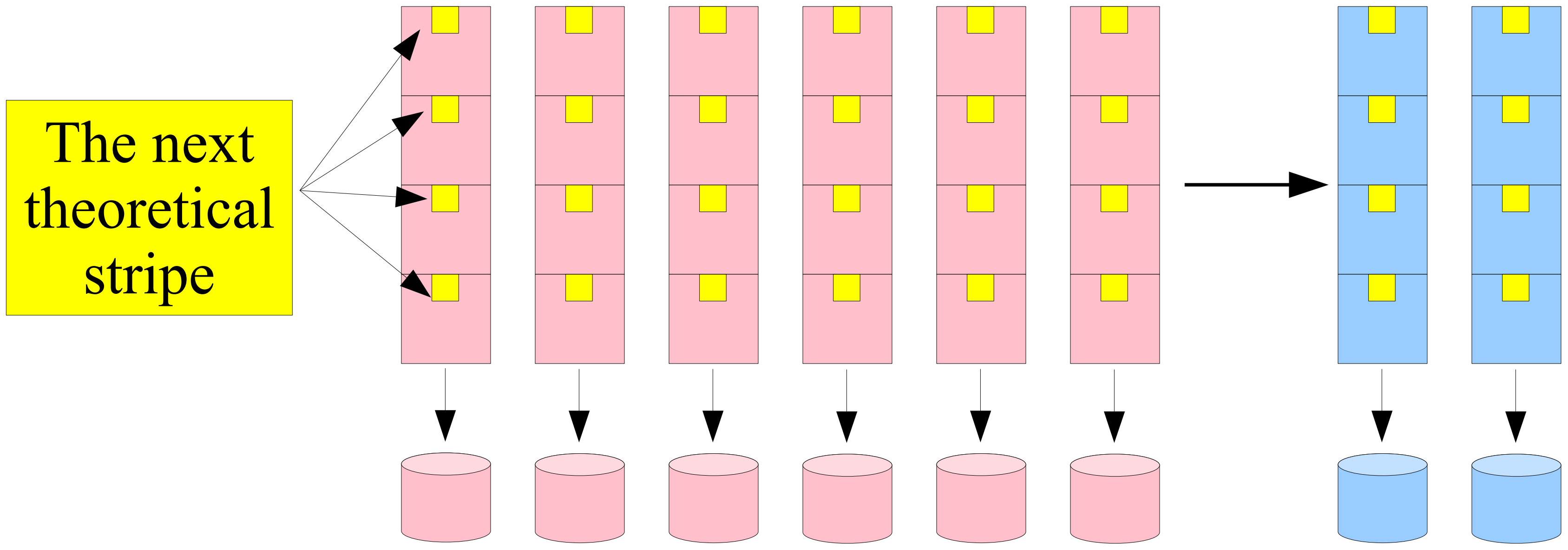
Two Views of a "Stripe"

- The Systems View:
  - The minimum partition of the system that encodes and decodes together.
  - Groups together theoretical stripes for performance.



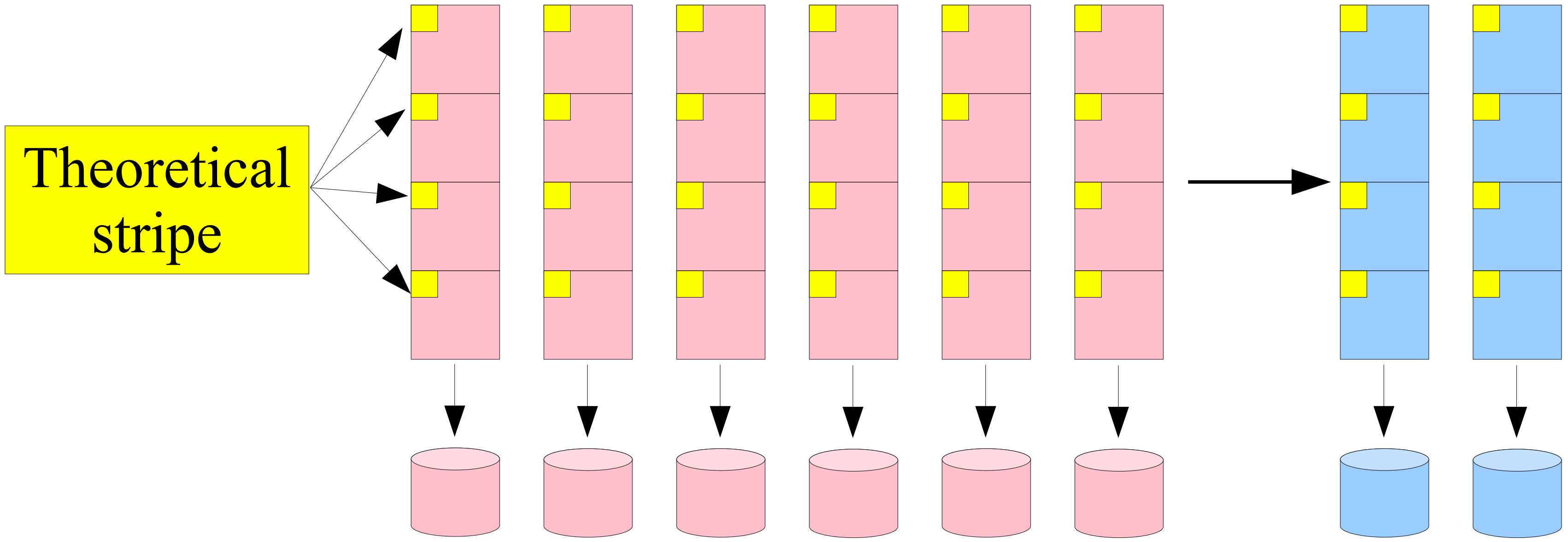
Two Views of a "Stripe"

- The Systems View:
  - The minimum partition of the system that encodes and decodes together.
  - Groups together theoretical stripes for performance.



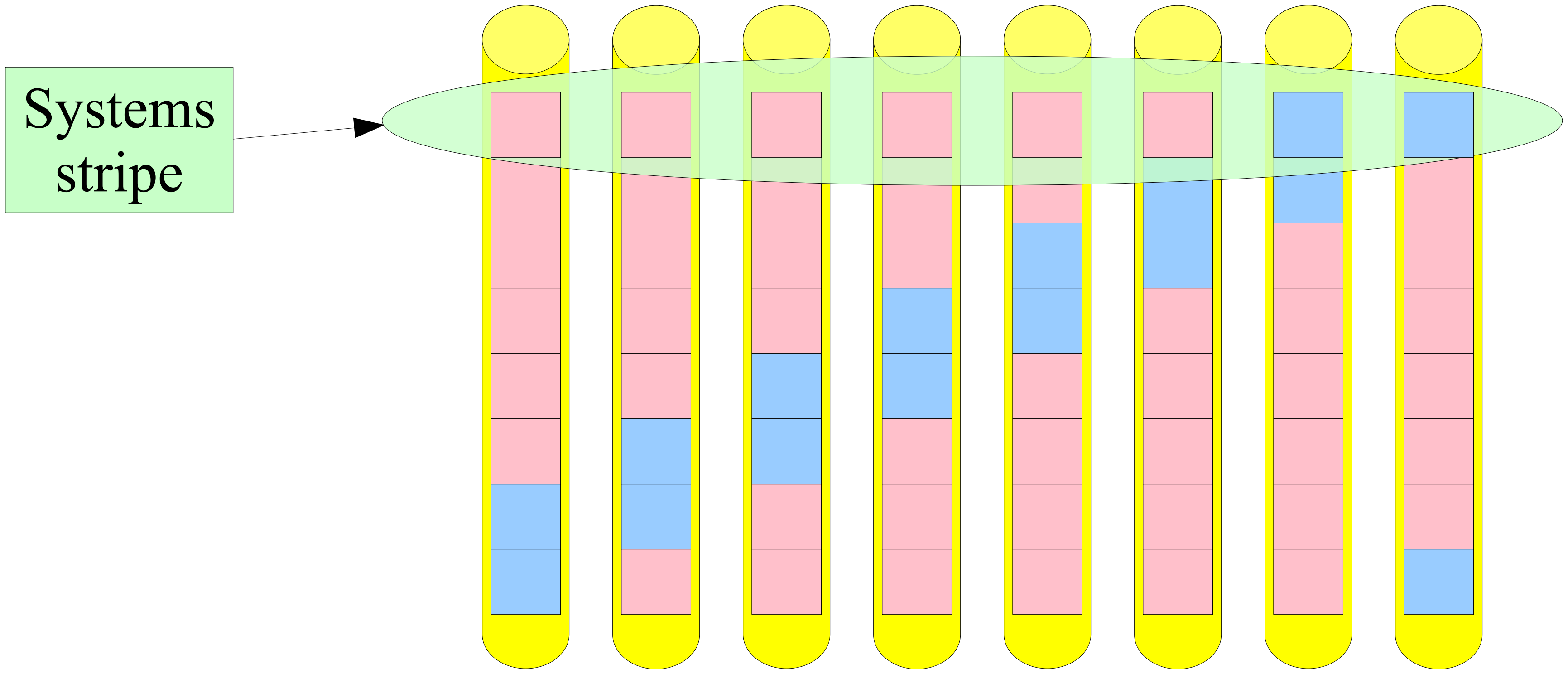
Two Views of a "Stripe"

- Why the two views?
  - Because *w* is small (often 1 bit), and operating over regions of symbols is much faster than operating over single symbols (Think XOR).



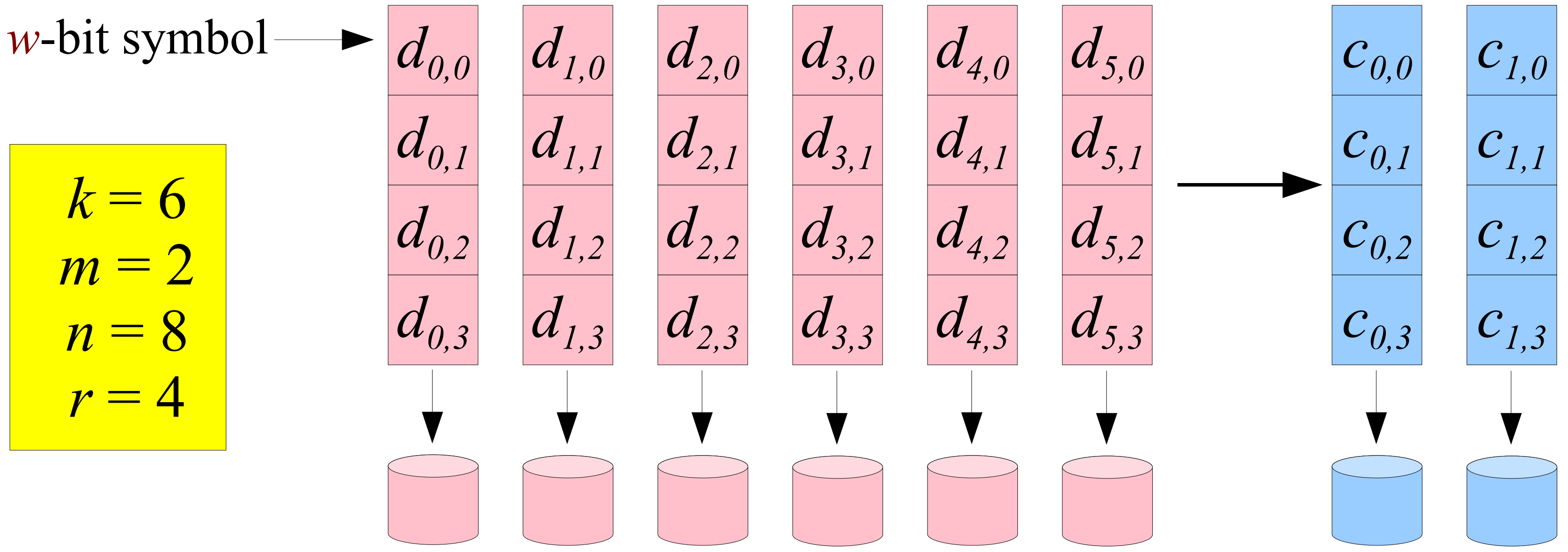
Two Views of a "Stripe"

- Why the two views?
  - And you can balance load by partitioning the system into many systems stripes and rotating ids.



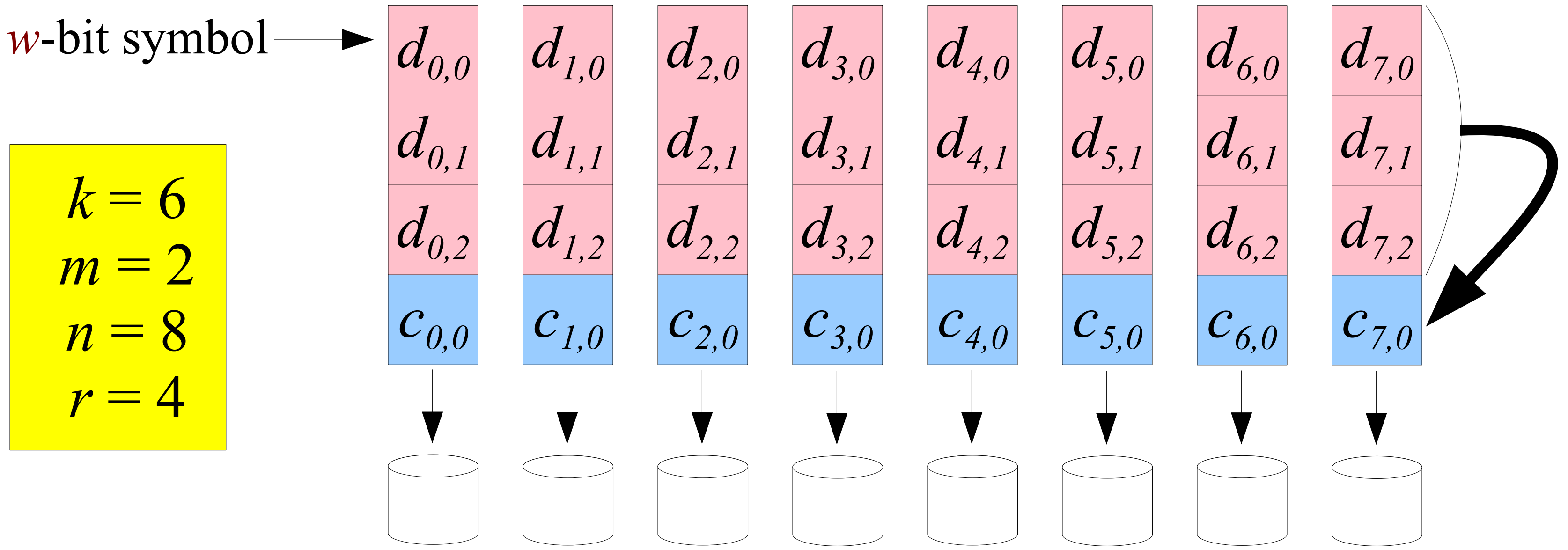
For the rest of this tutorial

- We will express erasure codes using theoretical stripes.
- We have *kr* symbols of data, each of which is *w* bits, and our disk system holds *nr* symbols of data+coding.



w-bit symbol
$k = 6$
$m = 2$
$n = 8$
$r = 4$
$d_{0,0}$ $d_{1,0}$ $d_{2,0}$ $d_{3,0}$ $d_{4,0}$ $d_{5,0}$
$d_{0,1}$ $d_{1,1}$ $d_{2,1}$ $d_{3,1}$ $d_{4,1}$ $d_{5,1}$
$d_{0,2}$ $d_{1,2}$ $d_{2,2}$ $d_{3,2}$ $d_{4,2}$ $d_{5,2}$
$d_{0,3}$ $d_{1,3}$ $d_{2,3}$ $d_{3,3}$ $d_{4,3}$ $d_{5,3}$
$c_{0,0}$ $c_{1,0}$
$c_{0,1}$ $c_{1,1}$
$c_{0,2}$ $c_{1,2}$
$c_{0,3}$ $c_{1,3}$

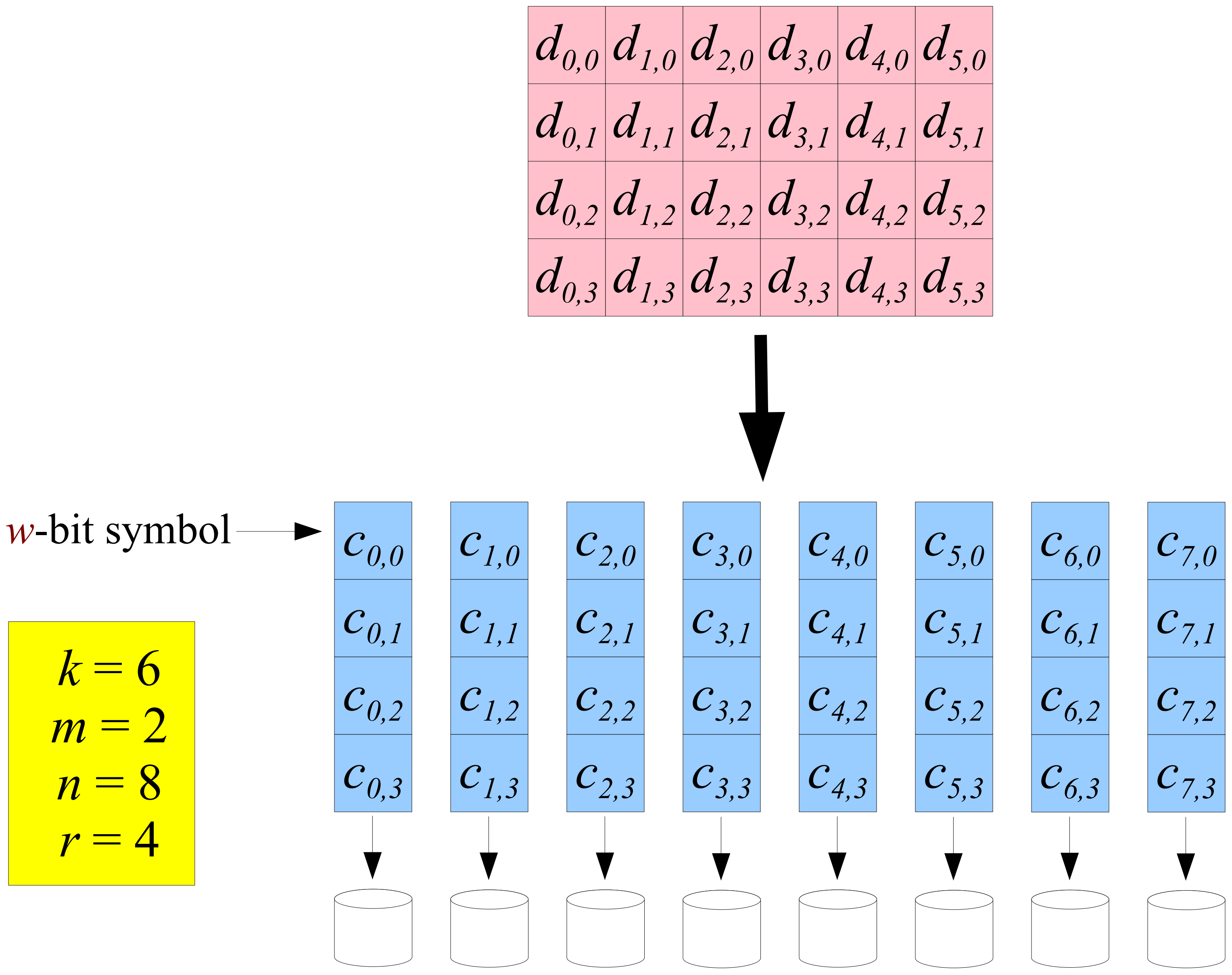
Systematic, Horizontal Code



w-bit symbol
$d_{0,0}$ $d_{1,0}$ $d_{2,0}$ $d_{3,0}$ $d_{4,0}$ $d_{5,0}$ $d_{6,0}$ $d_{7,0}$
$d_{0,1}$ $d_{1,1}$ $d_{2,1}$ $d_{3,1}$ $d_{4,1}$ $d_{5,1}$ $d_{6,1}$ $d_{7,1}$
$d_{0,2}$ $d_{1,2}$ $d_{2,2}$ $d_{3,2}$ $d_{4,2}$ $d_{5,2}$ $d_{6,2}$ $d_{7,2}$
$c_{0,0}$ $c_{1,0}$ $c_{2,0}$ $c_{3,0}$ $c_{4,0}$ $c_{5,0}$ $c_{6,0}$ $c_{7,0}$

$k = 6$
$m = 2$
$n = 8$
$r = 4$

Systematic, Vertical Code

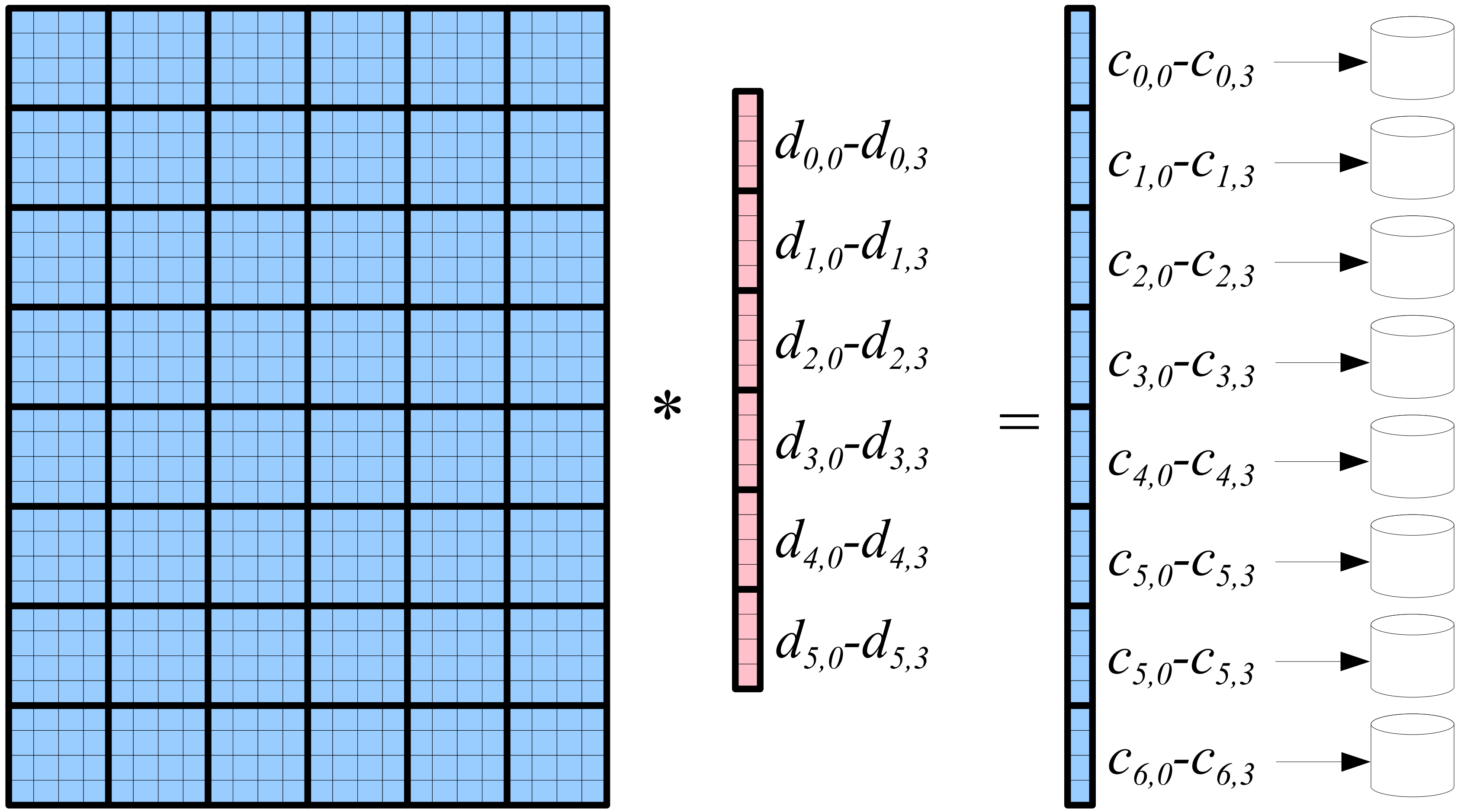

$d_{0,0}$ $d_{1,0}$ $d_{2,0}$ $d_{3,0}$ $d_{4,0}$ $d_{5,0}$
$d_{0,1}$ $d_{1,1}$ $d_{2,1}$ $d_{3,1}$ $d_{4,1}$ $d_{5,1}$
$d_{0,2}$ $d_{1,2}$ $d_{2,2}$ $d_{3,2}$ $d_{4,2}$ $d_{5,2}$
$d_{0,3}$ $d_{1,3}$ $d_{2,3}$ $d_{3,3}$ $d_{4,3}$ $d_{5,3}$
w-bit symbol
$c_{0,0}$ $c_{1,0}$ $c_{2,0}$ $c_{3,0}$ $c_{4,0}$ $c_{5,0}$ $c_{6,0}$ $c_{7,0}$
$c_{0,1}$ $c_{1,1}$ $c_{2,1}$ $c_{3,1}$ $c_{4,1}$ $c_{5,1}$ $c_{6,1}$ $c_{7,1}$
$c_{0,2}$ $c_{1,2}$ $c_{2,2}$ $c_{3,2}$ $c_{4,2}$ $c_{5,2}$ $c_{6,2}$ $c_{7,2}$
$c_{0,3}$ $c_{1,3}$ $c_{2,3}$ $c_{3,3}$ $c_{4,3}$ $c_{5,3}$ $c_{6,3}$ $c_{7,3}$
$k = 6$
$m = 2$
$n = 8$
$r = 4$

Non-systematic Code

The Basics: Generator Matrices

Expressing Codes with Generator Matrices

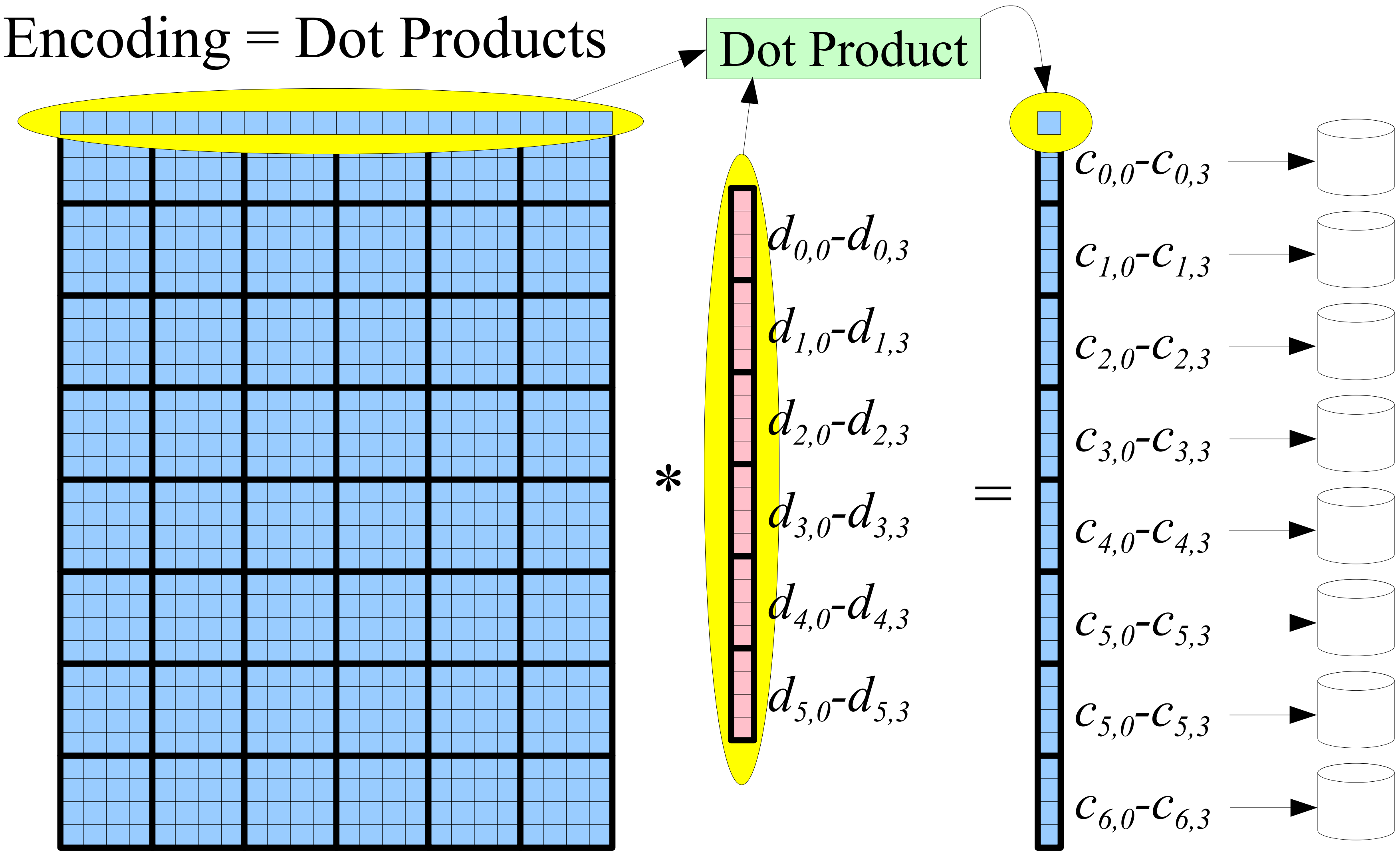
Non-systematic code with *k*=6, *n*=8, *r*=4.



Generator Matrix (G^T): *nr* X *kr* *w*-bit symbols

Expressing Codes with Generator Matrices

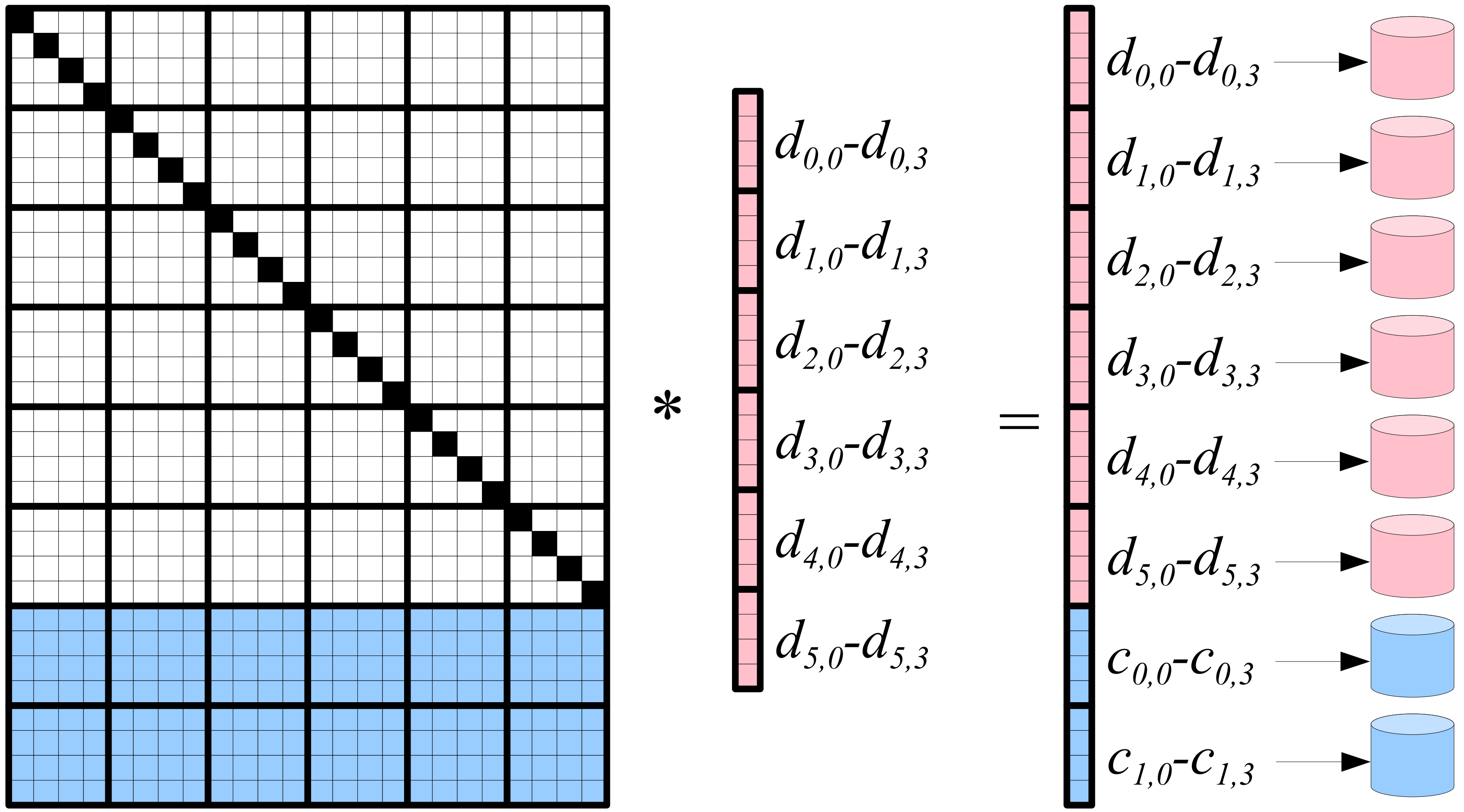
Encoding = Dot Products



Generator Matrix (G^T): nr X kr w-bit symbols

Expressing Codes with Generator Matrices

Systematic code with *k*=6, *n*=8, *r*=4.



Generator Matrix *(G^T)*: *nr* X *kr* *w*-bit symbols

Two concrete examples of encoding

Systematic code with k=6, n=8, r=1, w=8 (Linux RAID-6)

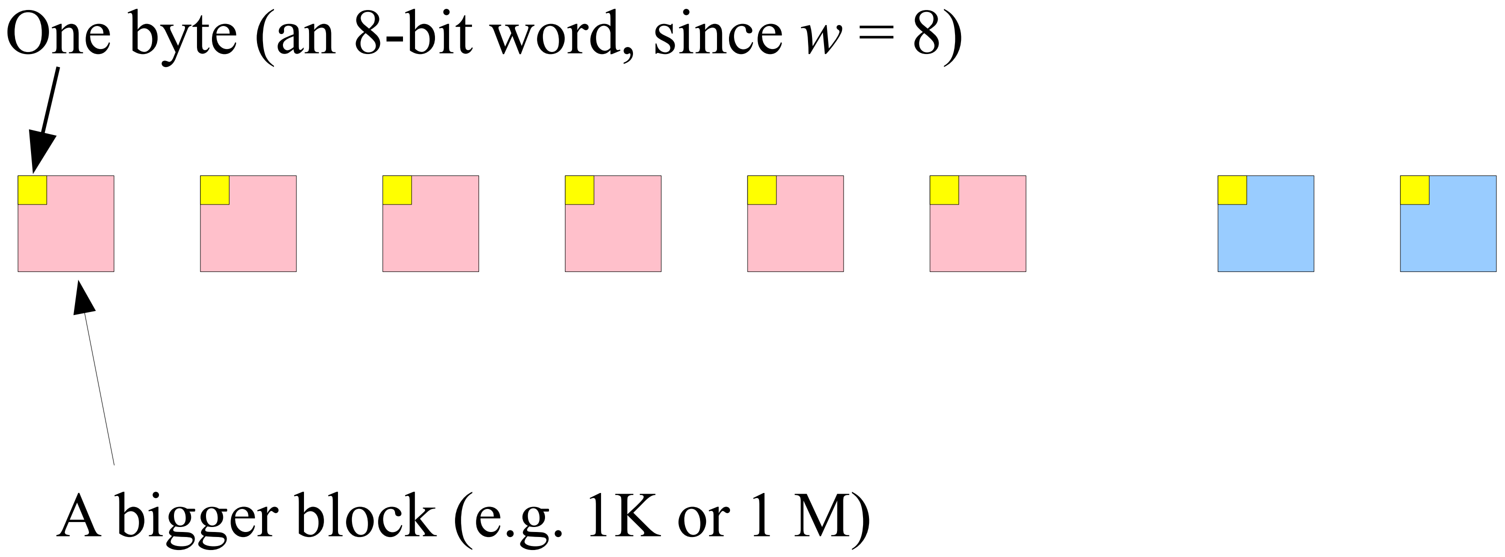
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 32 & 16 & 8 & 4 & 2 & 1 \end{bmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ p \\ q \end{bmatrix}$$

Generator Matrix (G^T)

Two concrete examples of encoding

Systematic code with *k*=6, *n*=8, *r*=1, *w*=8 (Linux RAID-6)

Let's map this to the systems view of a stripe:


One byte (an 8-bit word, since *w* = 8)
A bigger block (e.g. 1K or 1 M)

Two concrete examples of encoding

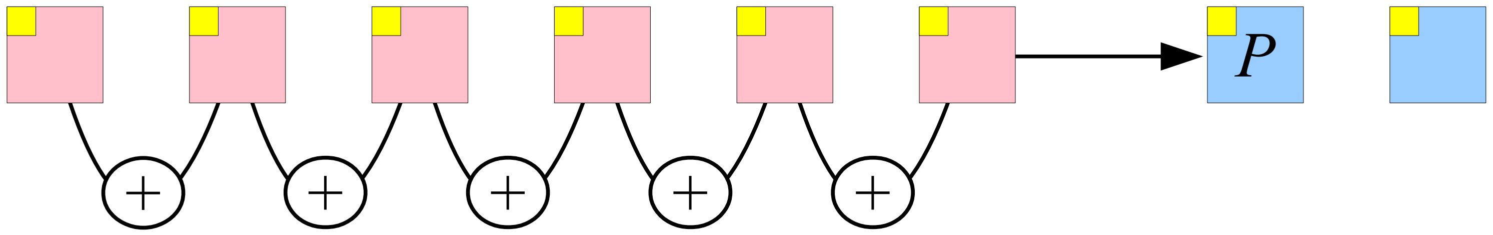
Systematic code with *k*=6, *n*=8, *r*=1, *w*=8 (Linux RAID-6)

Let's map this to the systems view of a stripe:

And calculate the first coding block (P):

1	1	1	1	1	1

P's row of the generator matrix



Addition = XOR. Can do that 64/128 bits at a time.

Two concrete examples of encoding

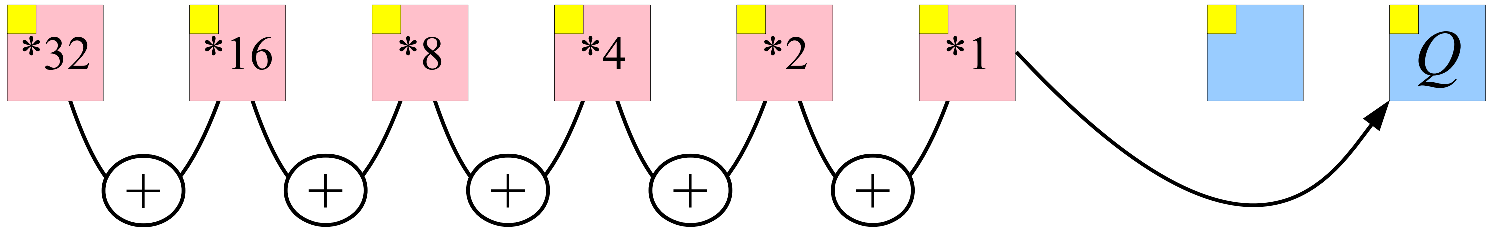
Systematic code with *k*=6, *n*=8, *r*=1, *w*=8 (Linux RAID-6)

Let's map this to the systems view of a stripe:

And calculate the second coding block (Q):

32	16	8	4	2	1

Q's row of the generator matrix

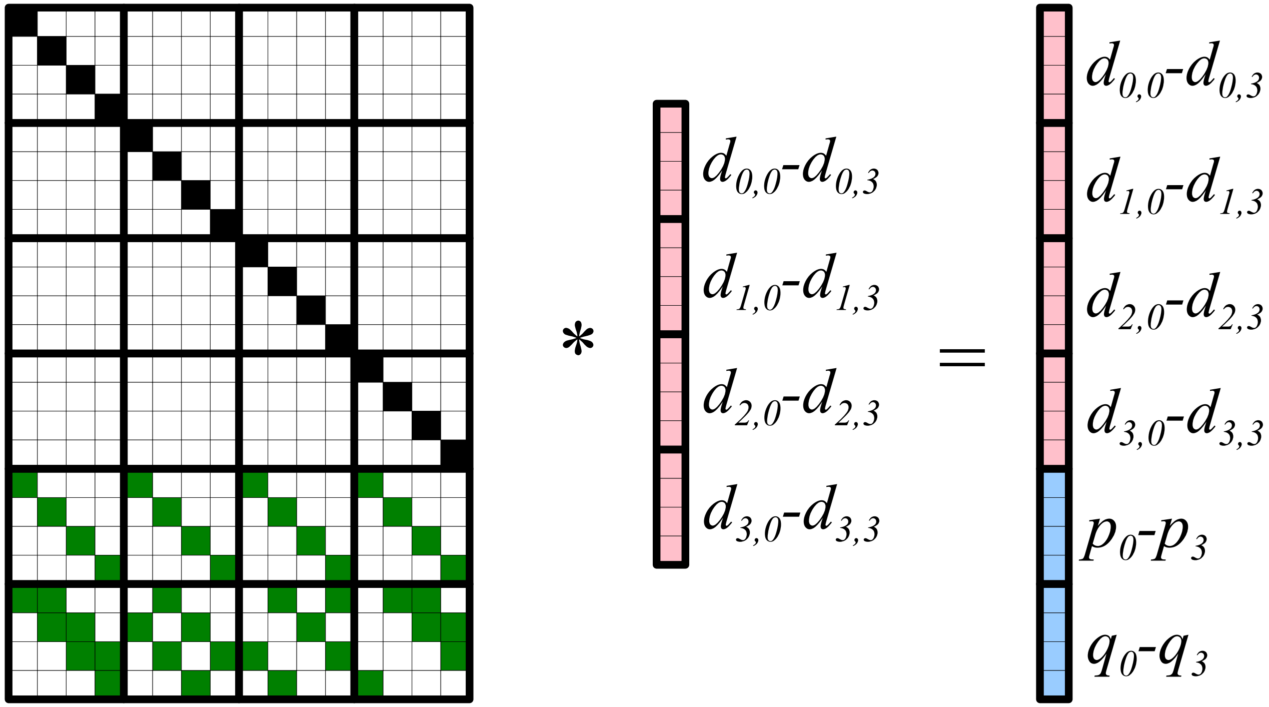


Addition still equals XOR.

Multiplication = "Galois Field Multiplication" *GF(2^w)*.
Complex, but still pretty fast (more on that later)

Two concrete examples of encoding

Systematic code with *k*=4, *n*=6, *r*=4, *w*=1 (RDP RAID-6)

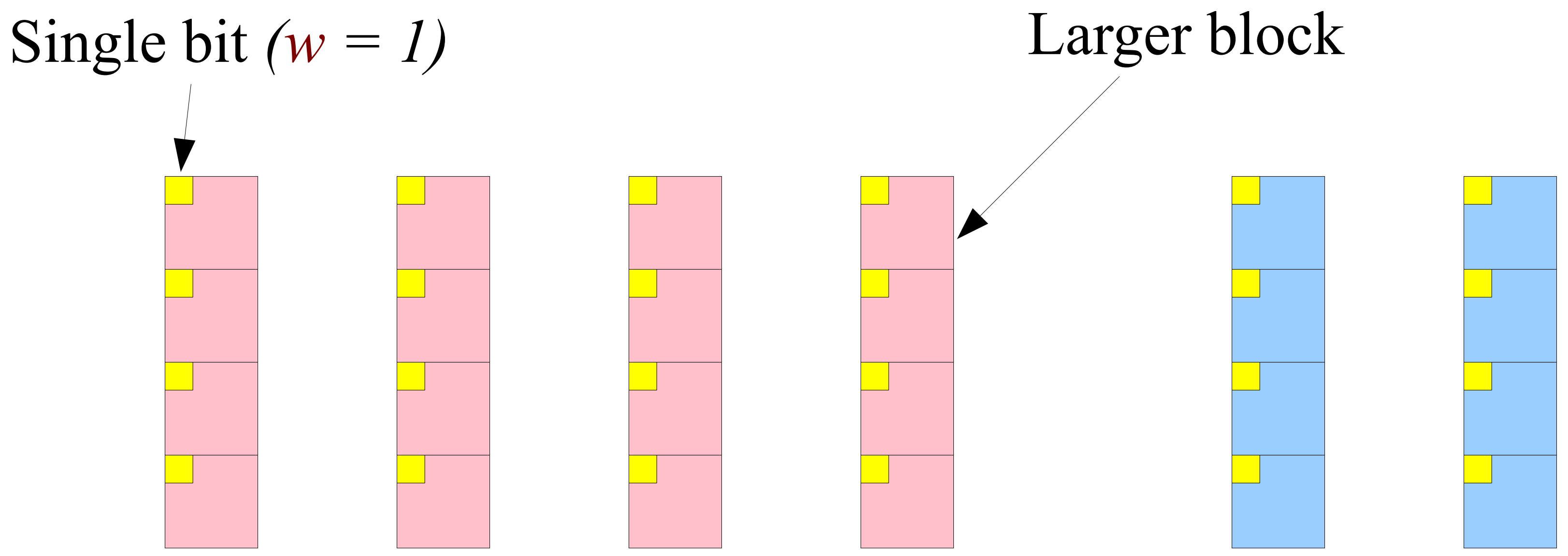


Generator Matrix (G^T):

Two concrete examples of encoding

Systematic code with *k*=4, *n*=6, *r*=4, *w*=1 (RDP RAID-6)

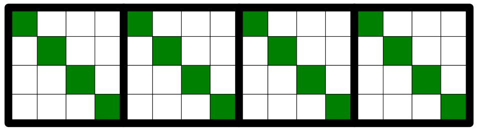
<u>Viewed as a systems stripe:</u>



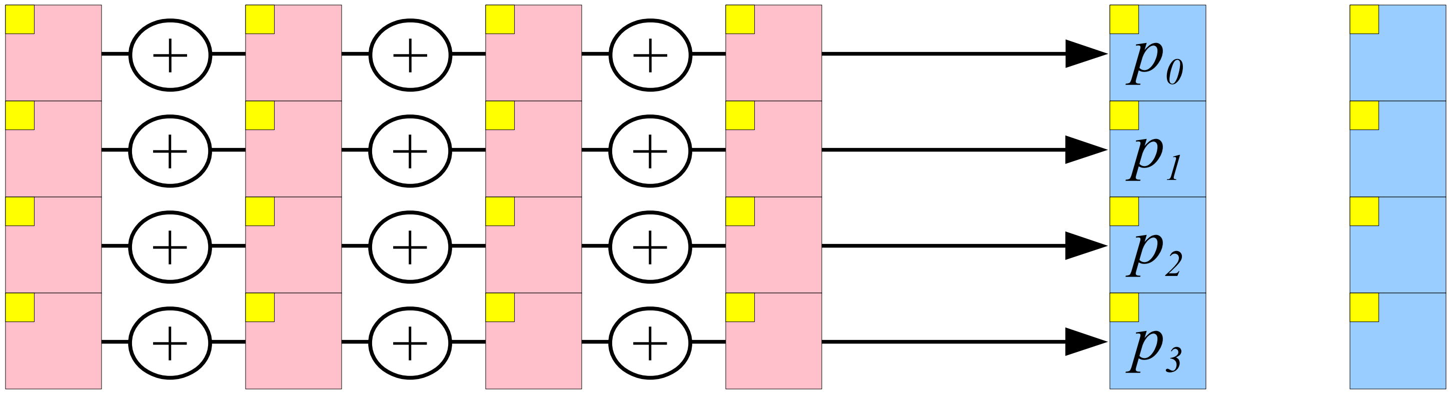
Two concrete examples of encoding

Systematic code with k=4, n=6, r=4, w=1 (RDP RAID-6)

Calculating the first coding block (P):



$p_0 - p_3$'s rows of the generator matrix

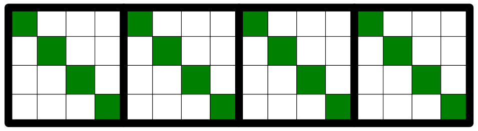


Again, we get to XOR blocks of bytes instead of single bits.

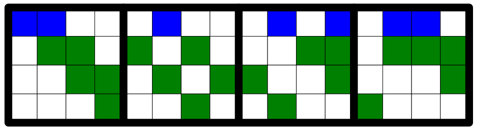
Two concrete examples of encoding

Systematic code with *k*=4, *n*=6, *r*=4, *w*=1 (RDP RAID-6)

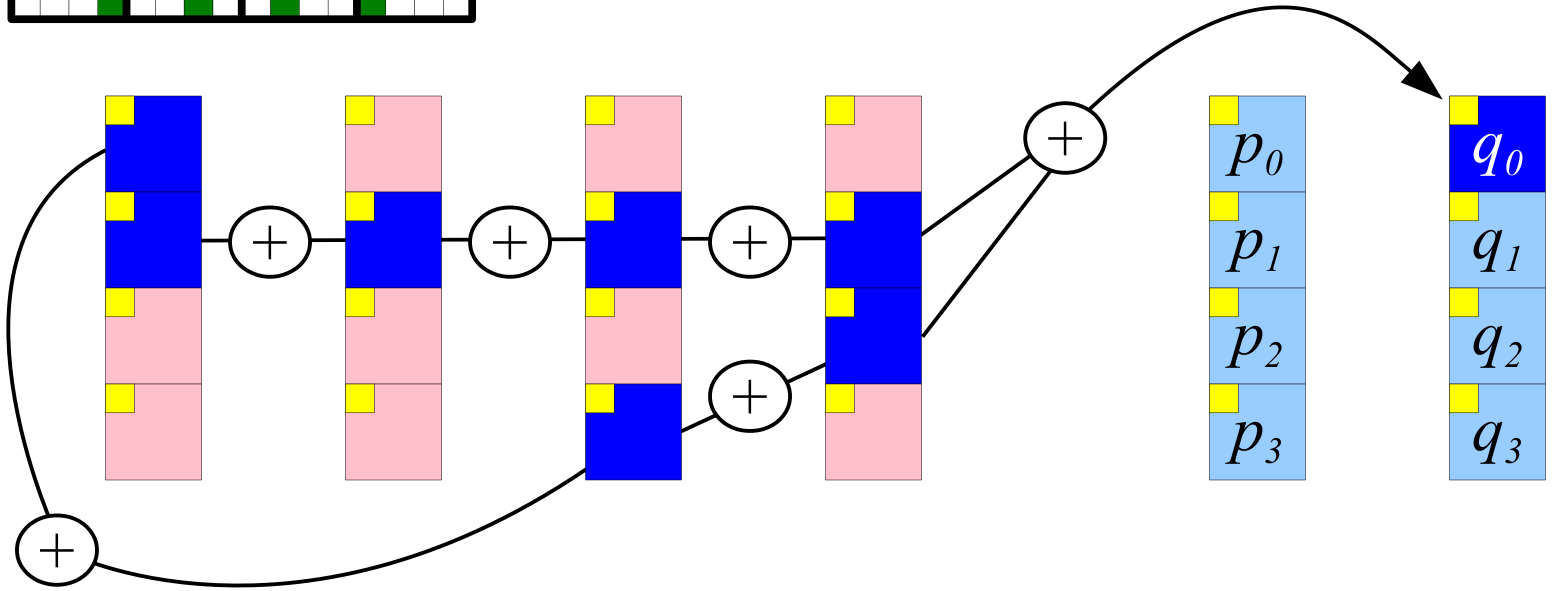
Calculating the second coding block (Q): - Just q_0 for now.



$p_0 - p_3$'s rows of the generator matrix



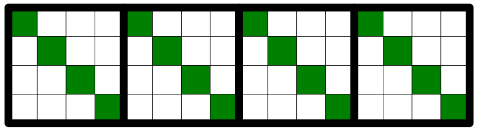
$q_0 - q_3$'s rows of the generator matrix



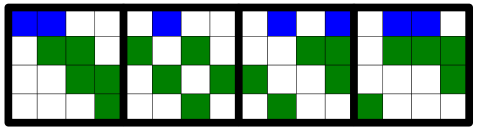
Two concrete examples of encoding

Systematic code with *k*=4, *n*=6, *r*=4, *w*=1 (RDP RAID-6)

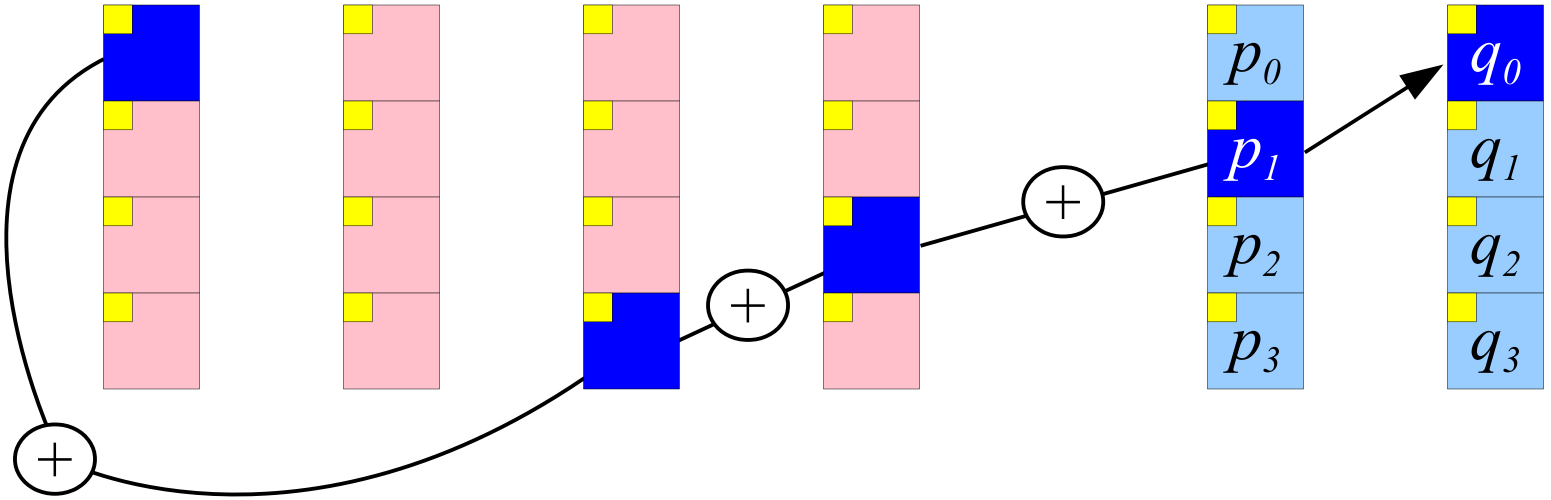
Calculating the second coding block (Q): - Just q_0 for now.



$p_0 - p_3$'s rows of the generator matrix



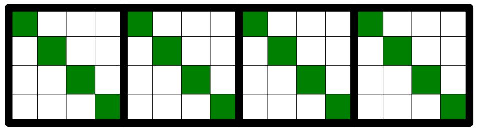
$q_0 - q_3$'s rows of the generator matrix



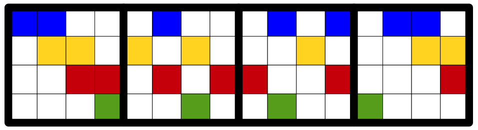
Two concrete examples of encoding

Systematic code with *k*=4, *n*=6, *r*=4, *w*=1 (RDP RAID-6)

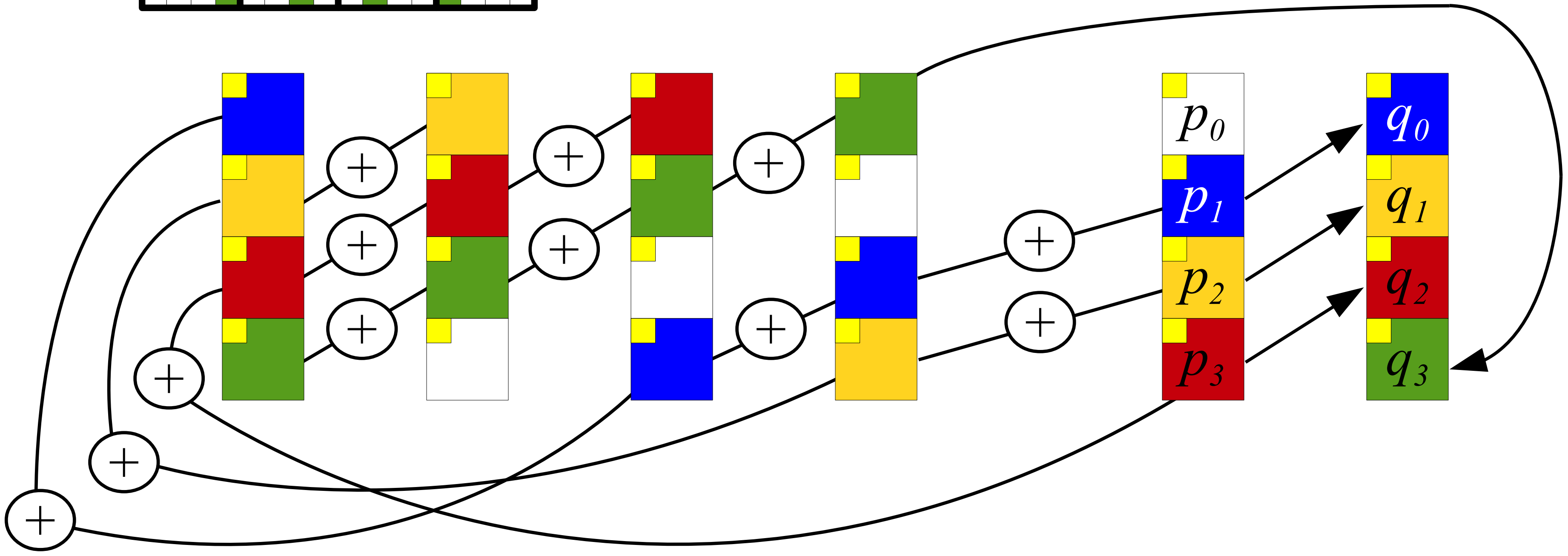
Calculating the second coding block (Q):



$p_0 - p_3$'s rows of the generator matrix



$q_0 - q_3$'s rows of the generator matrix



Arithmetic for Erasure Codes

- When $w = 1$: XOR's only.
- Otherwise, Galois Field Arithmetic $GF(2^w)$
  - w is 2, 4, 8, 16, 32, 64, 128 so that words fit evenly into computer words.
  - Addition is equal to XOR.

    Nice because addition equals subtraction.
  - Multiplication is more complicated:
    - Gets more expensive as w grows.
    - Buffer-constant different from $a * b$.
    - Buffer * 2 can be done really fast.
    - Open source library support.

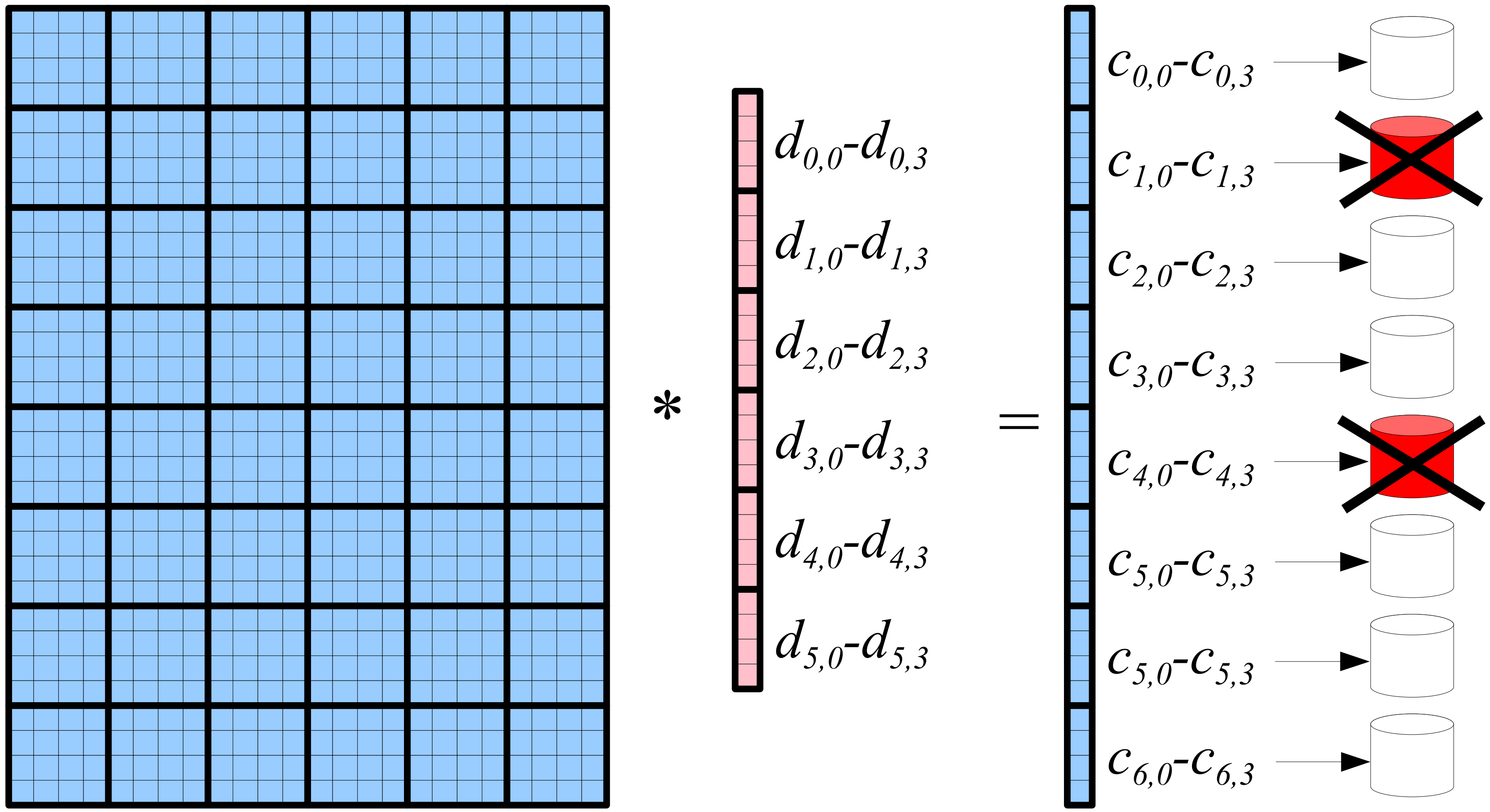
Arithmetic for Erasure Codes

- Multiplication of $a*b$ in $GF(2^w)$
  - Table lookup for $w = 2, 4 ,8$.
  - Three table lookups + a little arithmetic for $w = 16$
  - 16 table lookups + XORs for $w = 32$
  - More expensive for $w = 64, 128$.

- Multiplication of a buffer by a in $GF(2^w)$
  - Used to be roughly a factor of 4 worse than XOR for
    $w = 2, 4, 8, 16$.
  - SIMD instructions make those cache limited (~ 8 GB/s).
  - See my talk on Friday for more information.
  - $w = 32$ slightly slower. Others slower still.

Decoding with Generator Matrices

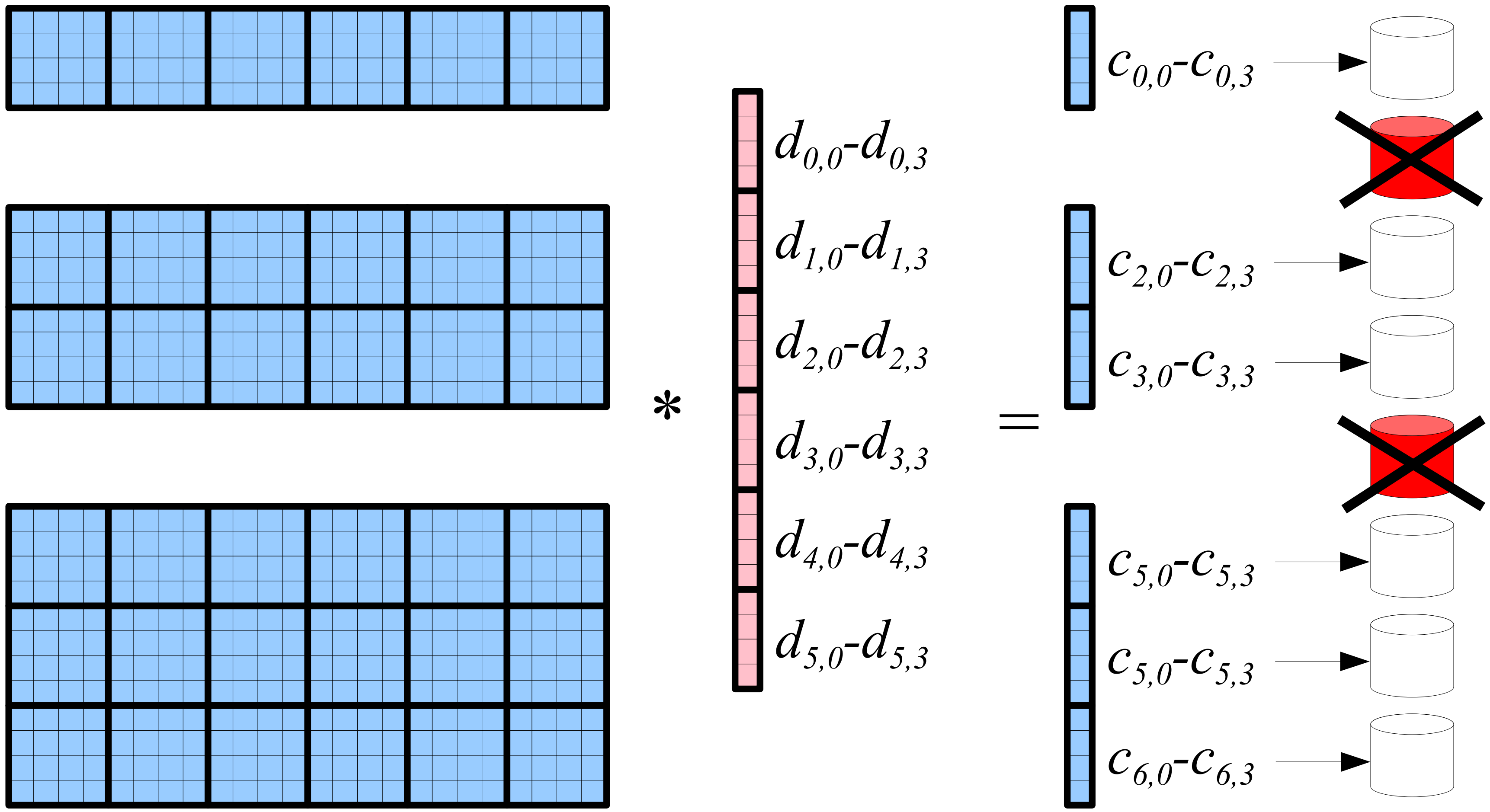
(Non-systematic code with *k*=6, *n*=8, *r*=4.)



Generator Matrix (G^T)

Decoding with Generator Matrices

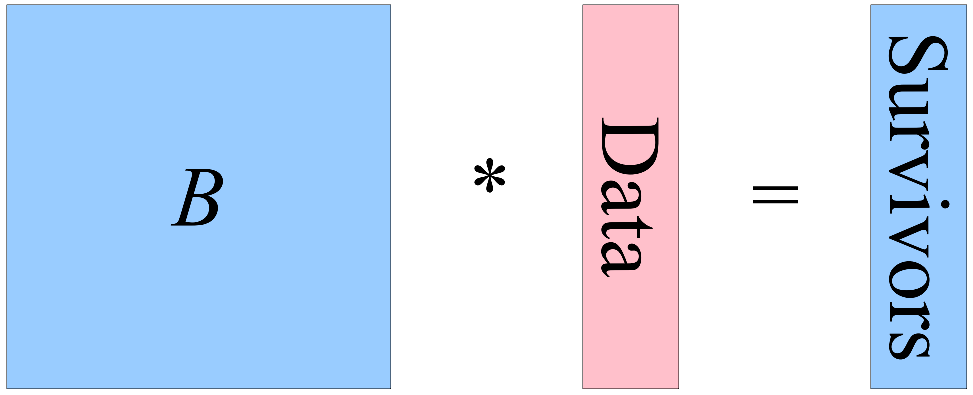
Step #1: Delete rows that correspond to failed disks.



Generator Matrix *(G^T)*

Decoding with Generator Matrices

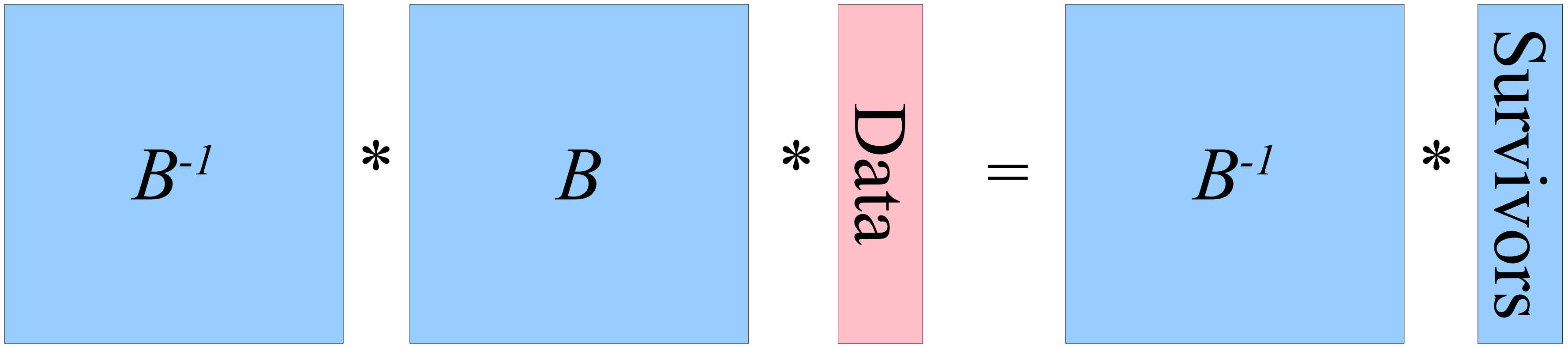
Step #2: Let's rewrite this equation so that looks like math.



Generator Matrix *(G^T)* with deleted rows

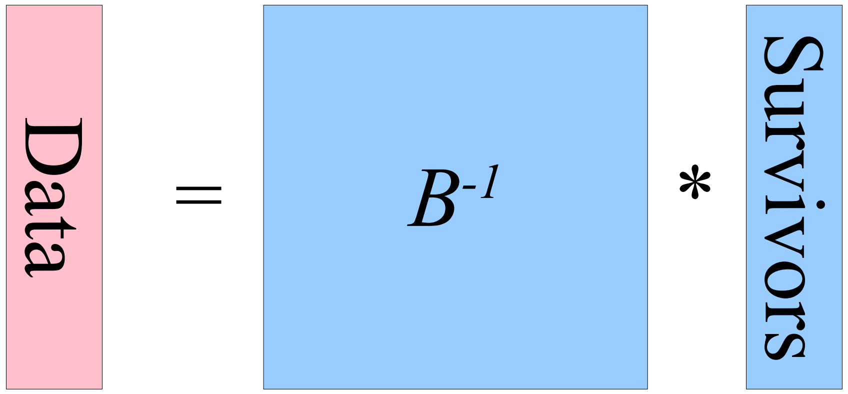
Decoding with Generator Matrices

Step #3: Invert *B* and multiply it by both sides of the equation:



Decoding with Generator Matrices

Voila – you now know how to decode the data!



What have we just learned?

- What a generator matrix is and how it encodes.
  - Systematic and non-systematic versions.
  - *nr* x *kr* matrix of *w*-bit words.
  - Encoding is done by dot products of rows of the generator with the data.
- XORs only when $w = 1$
- Otherwise use Galois Field arithmetic $GF(2^w)$
  - Addition = XOR
  - Multiplication uses special libraries.
- Decoding
  - Delete rows of the generator and then invert!

Code #1: Reed-Solomon Codes

(1960)

Reed-Solomon Codes: Properties and Constraints

- MDS Erasure codes for any n and k.
  - That means any $m = (n-k)$ failures can be tolerated without data loss.
- $r = 1$

  (Theoretical): One word per disk per stripe.
- w constrained so that $n \leq 2^w$.
- Systematic and non-systematic forms.

Systematic Reed-Solomon Codes

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	71	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$

Example with
$k = 6$ and $n = 9$.

- Create two sets X and Y.
- X has m elements: x_0 to x_{m-1}.
- Y has k elements: y_0 to y_{k-1}.
- Elements in $(X \cup Y)$ are distinct.
- $a_{i,j} = 1/(x_i + y_j)$ in $GF(2^w)$.

$X = \{ 1, 2, 3 \}$

$Y = \{ 4, 5, 6, 7, 8, 9 \}$

$$a_{1,2} = 1/(2+6) \text{ in } GF(2^8).$$
$$= 1/4 = 71.$$

(Using a Cauchy generator matrix)

Systematic Reed-Solomon Codes

$$\begin{array}{|c|c|c|c|c|c|}
\hline
1 & 0 & 0 & 0 & 0 & 0 \\ \hline
0 & 1 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 1 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 1 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & 1 & 0 \\ \hline
0 & 0 & 0 & 0 & 0 & 1 \\ \hline
a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} \\ \hline
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ \hline
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ \hline
\end{array}$$

Example with
$k = 6$ and $n = 9$.

Has the property that any ($k \times k$) submatrix is invertible (MDS).

(Using a Cauchy generator matrix)

Convenient property of Generators:

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1
1	1	1	1	1	1
1	🟦	🟦	🟦	🟦	🟦
1	🟦	🟦	🟦	🟦	🟦

Example with
$k = 6$ and $n = 9$.

If a matrix is MDS, then
you may transform it to another
MDS matrix by:

- Multiplying a row by a constant
- Multiplying a col by a constant
- Adding a row to another row
- Adding a column to another column.

Lets you transform the matrix so that it has nice properties.

(Also lets you create non-systematic codes)

Non-Systematic: Vandermonde Generator

$$\begin{bmatrix} 1^0 & 1^1 & 1^2 & 1^3 & 1^4 & 1^5 \\ 2^0 & 2^1 & 2^2 & 2^3 & 2^4 & 2^5 \\ 3^0 & 3^1 & 3^2 & 3^3 & 3^4 & 3^5 \\ 4^0 & 4^1 & 4^2 & 4^3 & 4^4 & 4^5 \\ 5^0 & 5^1 & 5^2 & 5^3 & 5^4 & 5^5 \\ 6^0 & 6^1 & 6^2 & 6^3 & 6^4 & 6^5 \\ 7^0 & 7^1 & 7^2 & 7^3 & 7^4 & 7^5 \\ 8^0 & 8^1 & 8^2 & 8^3 & 8^4 & 8^5 \\ 9^0 & 9^1 & 9^2 & 9^3 & 9^4 & 9^5 \end{bmatrix}$$

Example with
k = 6 and n = 9.

This is MDS,
so long as $n < 2^w$

Non-Systematic: Vandermonde Generator

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 & 1^5 \\ 2^0 & 2^1 & 2^2 & 2^3 & 2^4 & 2^5 \\ 3^0 & 3^1 & 3^2 & 3^3 & 3^4 & 3^5 \end{bmatrix}$$

Example with
$k = 6$ and $n = 9$.

The Vandermonde construction *only* applies to non-systematic codes.

This generator matrix is *not guaranteed to be MDS.*

So don't use it!

(You can convert an MDS non-systematic generator to a systematic one, though, if you care to.)

Code #2:
Cauchy Reed-Solomon Codes

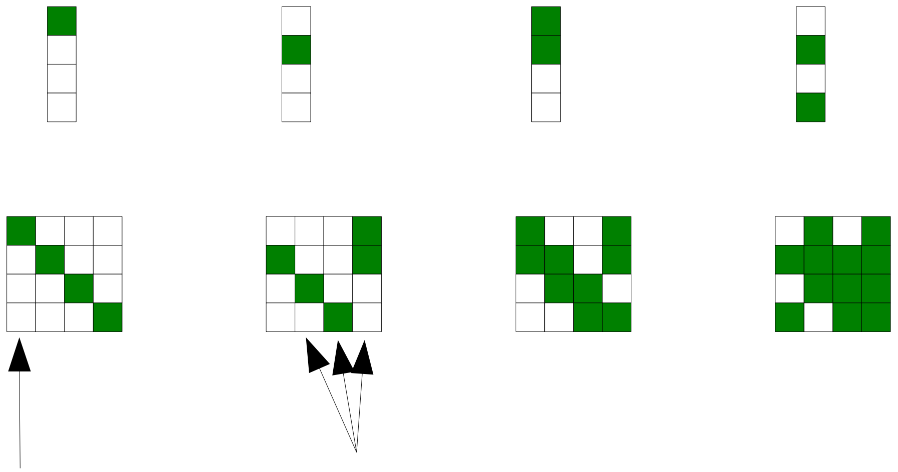
(1995)

Cauchy Reed Solomon Coding

Every number in *GF(2^w)* has three representations:

Examples in *GF(2^4)*

1. As a number — 1 — 2 — 3 — 10
2. As a *w*-element bit vector
3. As a *w* X *w* bit matrix



Column 0 = the bit vector representation

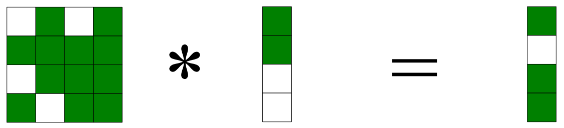
Column *i* = 2(column *i-1*) in *GF(2^w)*

Cauchy Reed Solomon Coding

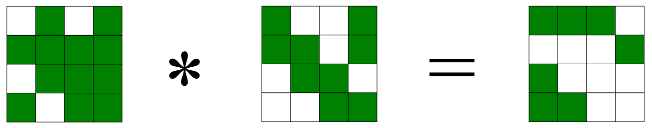
This has the great property that:

10 * 3 = 13

Matrix * vector = vector



Matrix * matrix = matrix



Examples in $GF(2^4)$

Cauchy Reed Solomon Coding

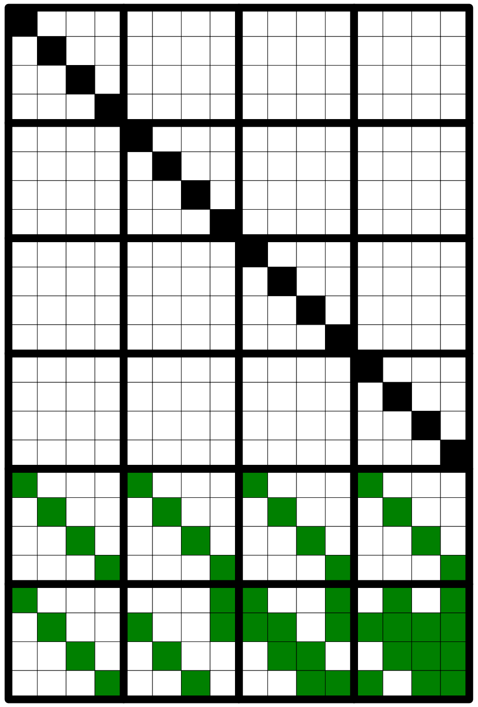
$k = 4$
$n = 6$
$r = 1$
$w = 4$

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	1	1	1
1	2	3	10

You may take a generator matrix for $w > 1$ and convert it to one with $w = 1$ and $r = rw$, by using the matrix representation of each element.

If one is MDS, then so is the other.

$k = 4$
$n = 6$
$r = 4$
$w = 1$



This allows you to perform only XOR operations.

Bottom Line for these Two Codes

General Purpose, MDS codes for any k and n:

- Reed-Solomon: $r = 1$, $n \leq 2^w$
  Uses Galois Field Arithmetic.

- Cauchy Reed-Solomon: $w = 1$, $n \leq 2^r$
  Uses only XOR's.

Methodology allows you to understand a rich variety of erasure codes (some of which you'll see later in this tutorial).

Theoretical Interlude

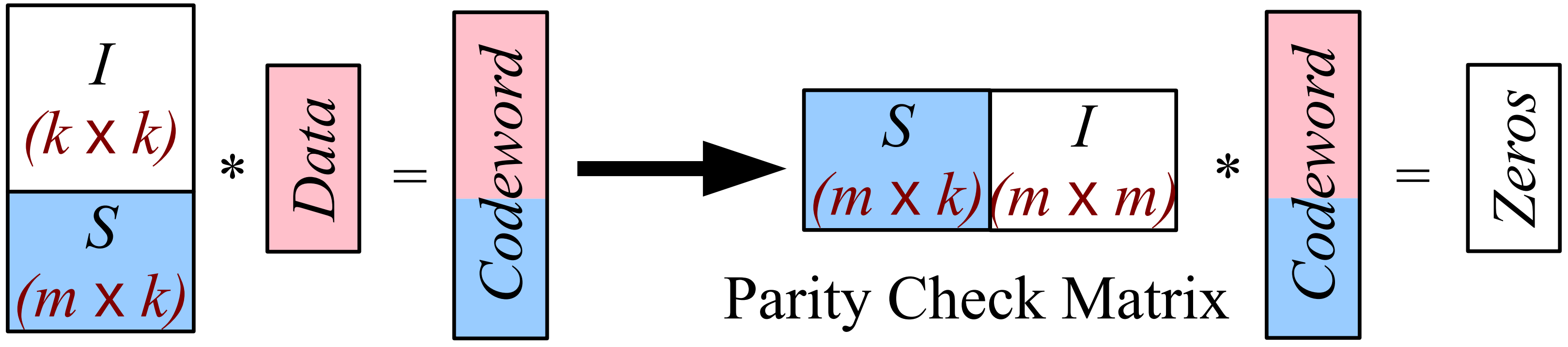
Information Theorists use lingo that is slightly different from what I have just described.

1. My "data" vector is called "the message."
2. What I call "data words," they call "message symbols."
3. The product vector is called "the codeword."
   In a systematic code, this is composed of data and coding words.
4. My generator matrix is the transpose of theirs.
5. Their message and codeword vectors are row vectors rather than column vectors. (Their pictures are hideous).

An Alternate Code Spec

A Parity Check Matrix is an $n \times m$ matrix which, when multiplied by the codeword, equals a vector of zeros.

It is a simple matter to take a generator matrix for a systematic code and turn it into a parity check matrix:



Generator Matrix (G^T)

Decoding with a Parity Check Matrix

Turn the matrix-vector product into a series of equations.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 32 & 16 & 8 & 4 & 2 & 1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ p \\ q \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 \oplus p = 0$$

$$32d_0 \oplus 16d_1 \oplus 8d_2 \oplus 4d_3 \oplus 2d_4 \oplus d_5 \oplus q = 0$$

Decoding with a Parity Check Matrix

$$d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 \oplus p = 0$$

$$32d_0 \oplus 16d_1 \oplus 8d_2 \oplus 4d_3 \oplus 2d_4 \oplus d_5 \oplus q = 0$$

Suppose d_1 and d_4 fail

#1: Put the failed words on the right of the equations

$$d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus p = d_1 \oplus d_4$$

$$32d_0 \oplus 8d_2 \oplus 4d_3 \oplus d_5 \oplus q = 16d_1 \oplus 2d_4$$

Decoding with a Parity Check Matrix

$$d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 \oplus p = 0$$

$$32d_0 \oplus 16d_1 \oplus 8d_2 \oplus 4d_3 \oplus 2d_4 \oplus d_5 \oplus q = 0$$

Suppose d_1 and d_4 fail

#2: Calculate the left sides, since those all exist.

$$d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus p = S_0 = d_1 \oplus d_4$$

$$32d_0 \oplus 8d_2 \oplus 4d_3 \oplus d_5 \oplus q = S_1 = 16d_1 \oplus 2d_4$$

Decoding with a Parity Check Matrix

$$\begin{array}{ccccccccccccccc} d_0 & \oplus & d_1 & \oplus & d_2 & \oplus & d_3 & \oplus & d_4 & \oplus & d_5 & \oplus & p & & = 0 \\ 32d_0 & \oplus & 16d_1 & \oplus & 8d_2 & \oplus & 4d_3 & \oplus & 2d_4 & \oplus & d_5 & \oplus & & q & = 0 \end{array}$$

Suppose d_1 and d_4 fail

#3: Solve using Gaussian Elimination or Matrix Inversion

$$\begin{array}{lcl} S_0 & = & d_1 \oplus d_4 \\ S_1 & = & 16d_1 \oplus 2d_4 \end{array} \longrightarrow \begin{array}{lcl} d_1 & = & \dfrac{(2S_0 \oplus S_1)}{(16 \oplus 2)} \\ d_4 & = & S_0 \oplus d_1 \end{array}$$

Decoding with a Parity Check Matrix

- Why bother with this?
  - It may be faster than the other method.
  - (Remember, multiplication by one (XOR) and two are very fast.)

$$d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus p = S_0 = d_1 \oplus d_4$$

$$32d_0 \oplus 8d_2 \oplus 4d_3 \oplus d_5 \oplus q = S_1 = 16d_1 \oplus 2d_4$$

Code #3: Linux RAID-6

(2002?)

Linux RAID-6

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 32 & 16 & 8 & 4 & 2 & 1 \end{bmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ p \\ q \end{bmatrix}$$

Generator Matrix (G^T)

$m = 2$
$r = 1$
$w = 8$ (typically)

$p_i = 1$
$q_i = 2^{k-i-1}$

MDS for $k < 256$

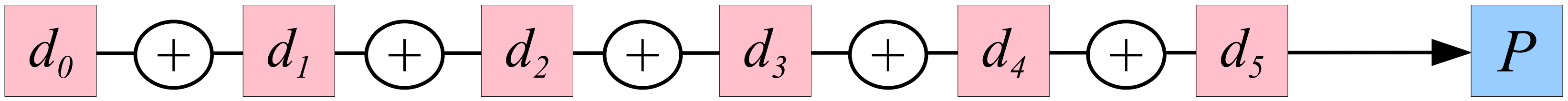
Is really fast at encoding/decoding. Why?

Linux RAID-6

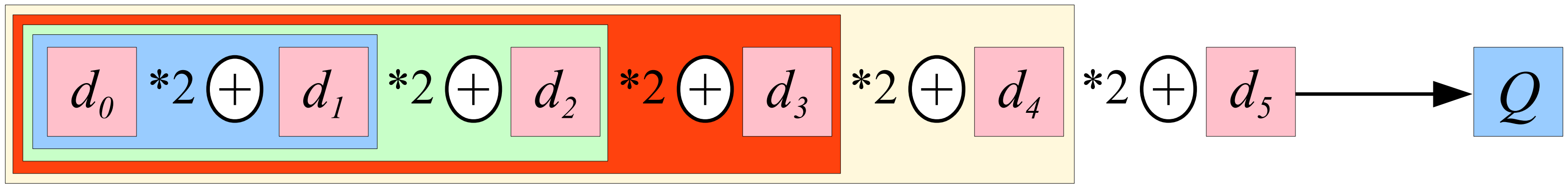
1	1	1	1	1	1
32	16	8	4	2	1

Why is encoding fast?

First, the *P* drive is simply parity.



Second, the *Q* drive leverages fast multiplication by two.



Linux RAID-6

To decode, use the Parity Check Matrix, and most of your multiplications are by one and two.

1	1	1	1	1	1	1	0
32	16	8	4	2	1	0	1

Very nicely structured code.

Code #4: EVENODD

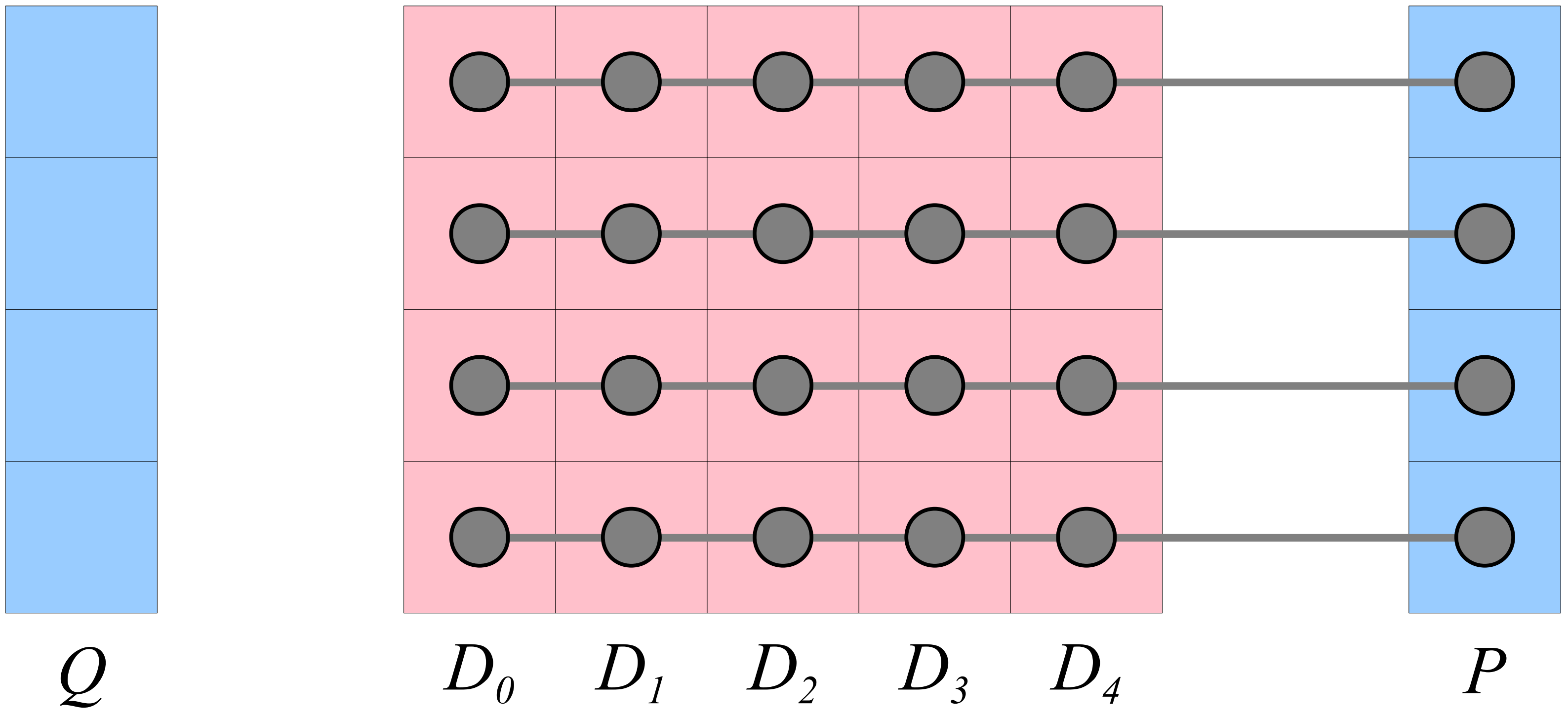
(1994)

EVENODD – Properties & Constraints

- RAID-6: $m = 2$
- Systematic
- Horizontal
- MDS
- $w = 1$: Only XOR operations.
- k must be a prime (that will be relaxed).
- r is set to k-1.

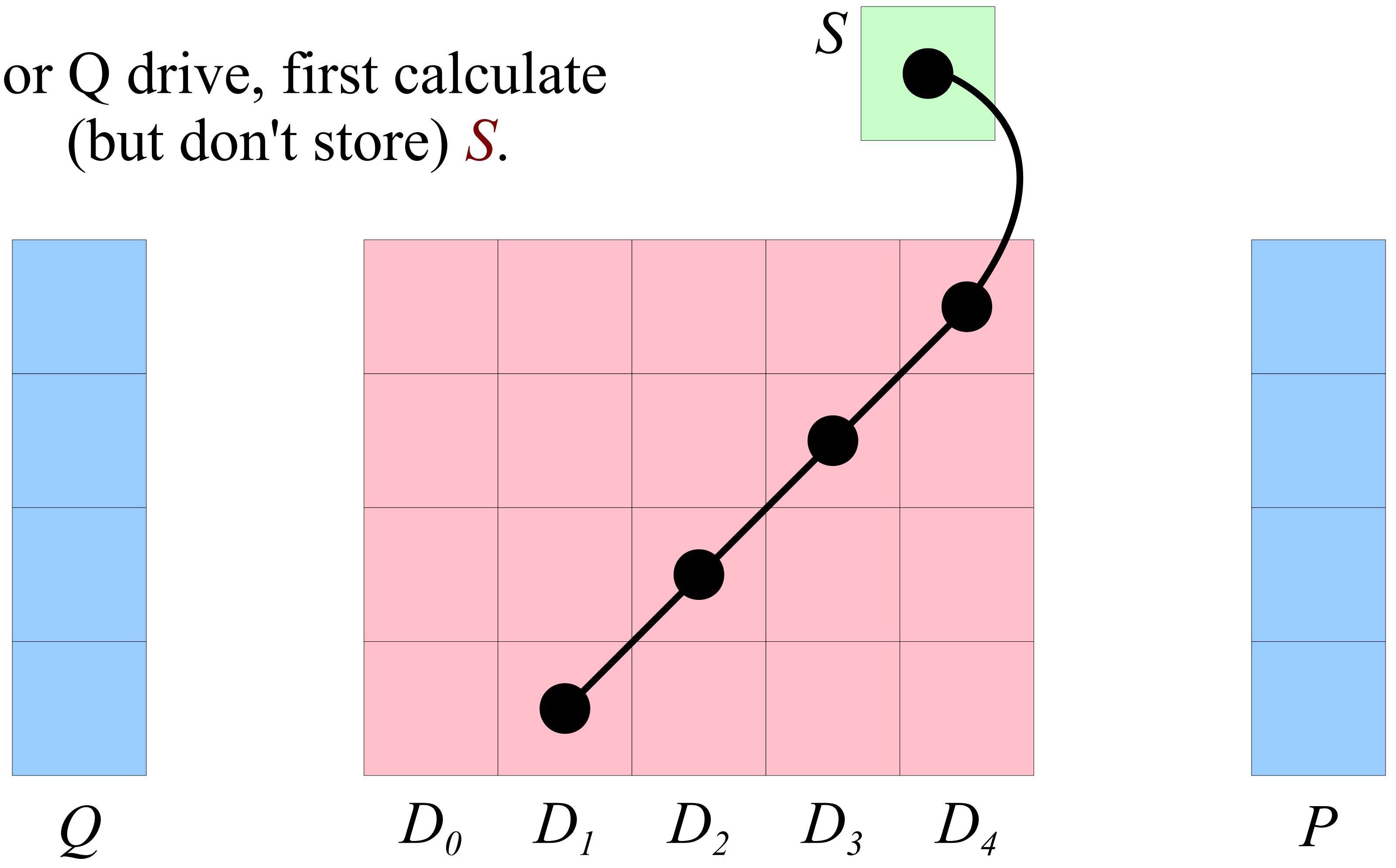
EVENODD Example: *k=5 (r=4)*

The *P* drive is parity (like in most codes).
Lines denote XOR relationship.



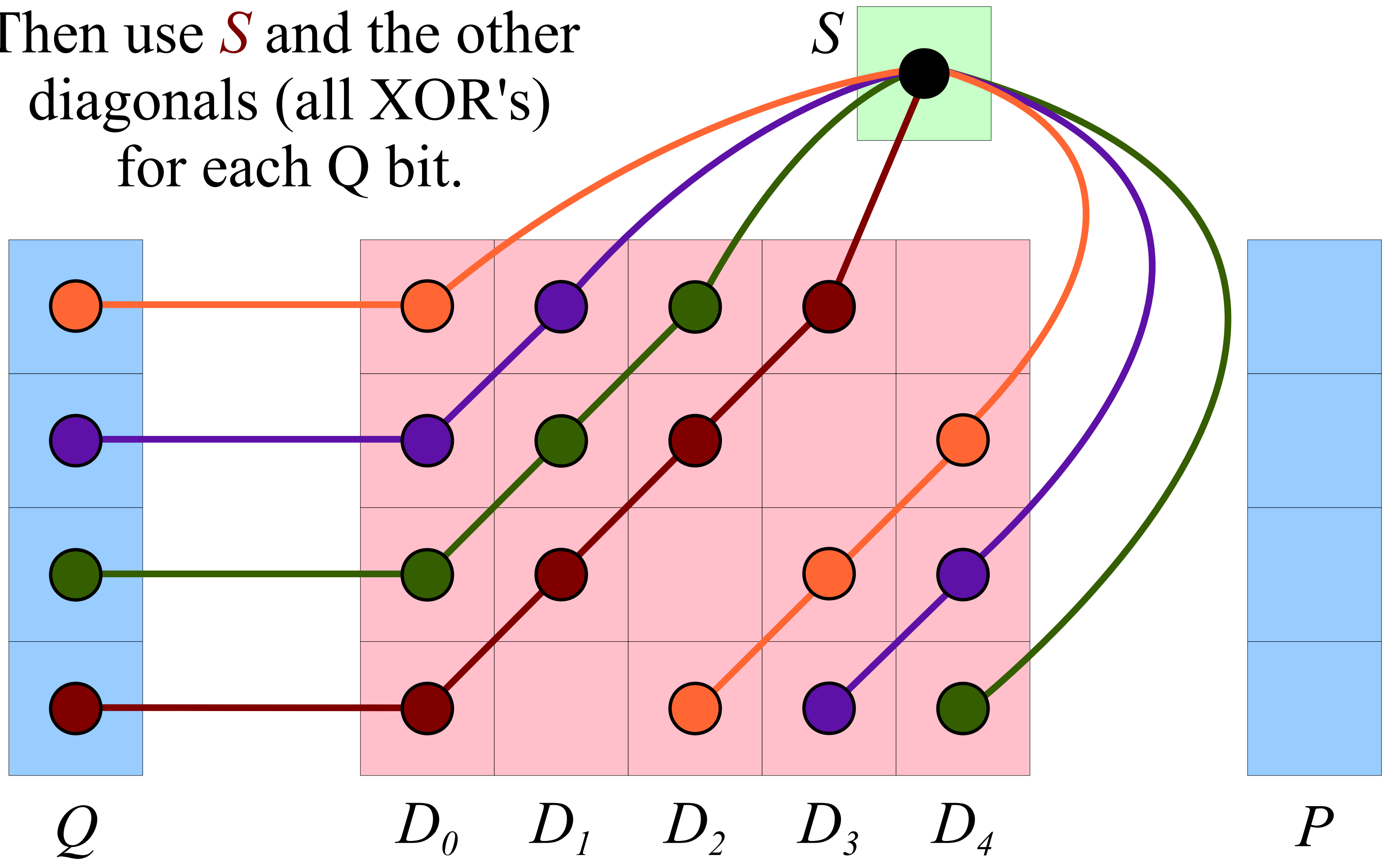
EVENODD Example: *k=5 (r=4)*

For Q drive, first calculate (but don't store) *S*.



EVENODD Example: *k=5 (r=4)*

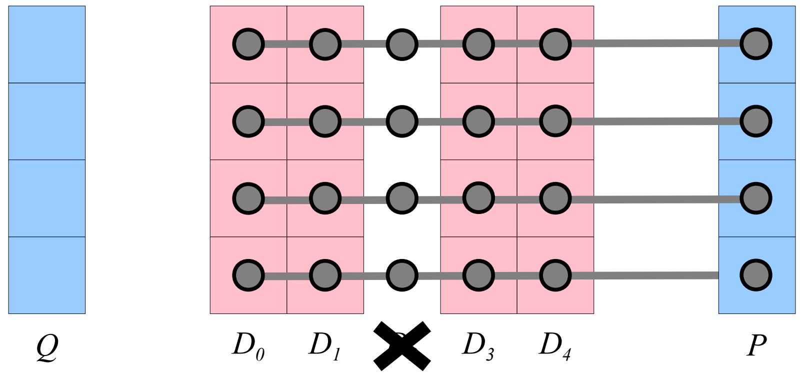
Then use *S* and the other diagonals (all XOR's) for each Q bit.



You'll note, each diagonal chain is missing one data drive.

EVENODD Example: *k=5 (r=4)*

Decoding single failures is simple: just use the P drive (more on that later with Cheng).

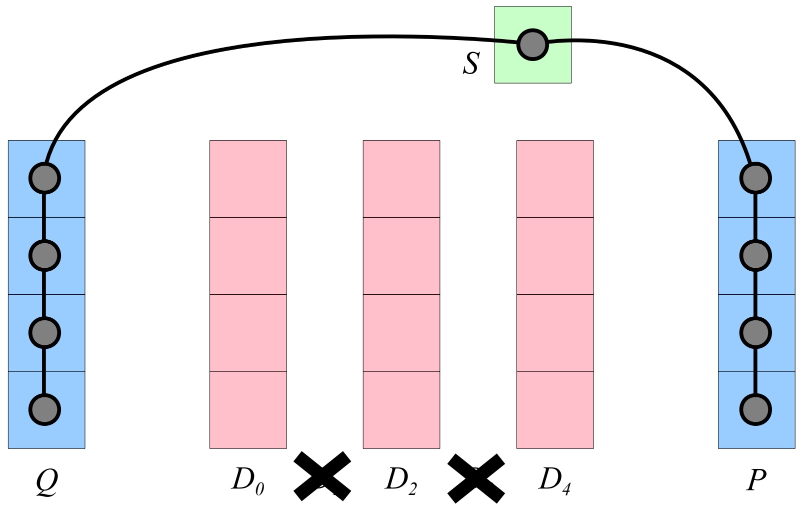


The difficult case is decoding when two data disks fail.

So, we'll go through an example when D_1 and D_3 fail.

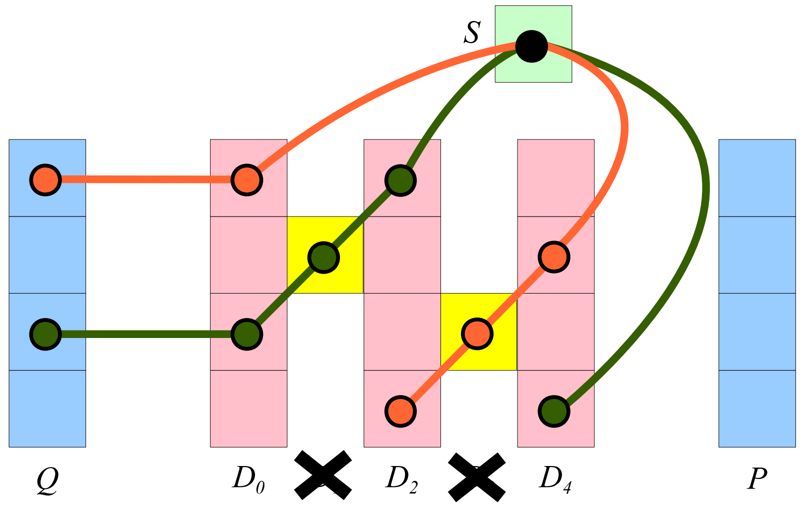
EVENODD Example: *k*=5 (*r*=4)

The first thing to do is calculate *S*, which equals the XOR of all of the bits in P and Q.



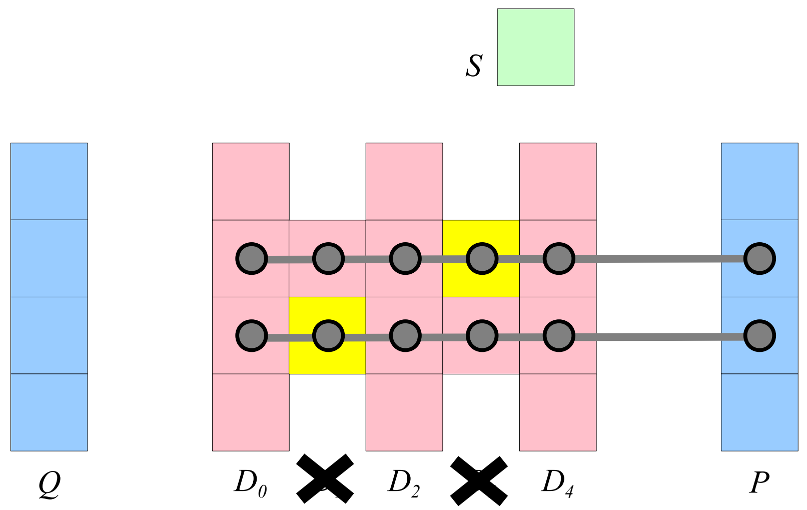
Then recall that each diagonal is missing a drive.

Use the ones missing D_1 and D_3 to recover one bit in D_3 and D_1.

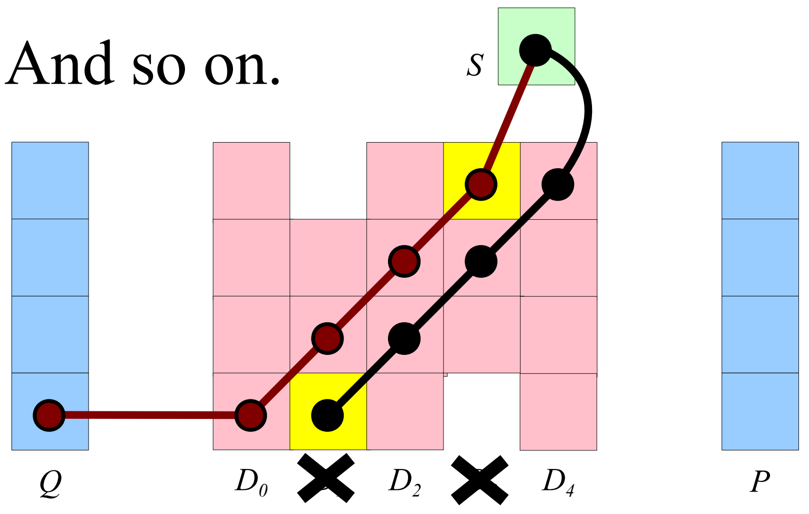


EVENODD Example: *k=5 (r=4)*

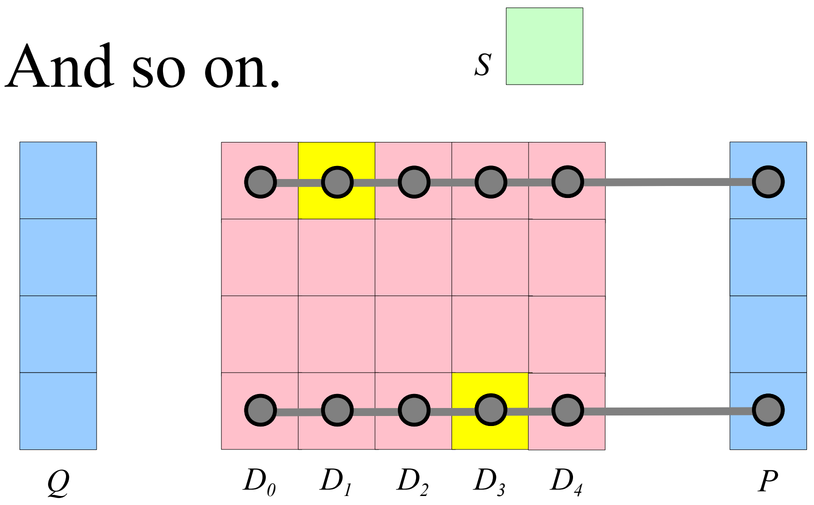
Now use these new bits and the P drive to decode two more bits.



And so on.

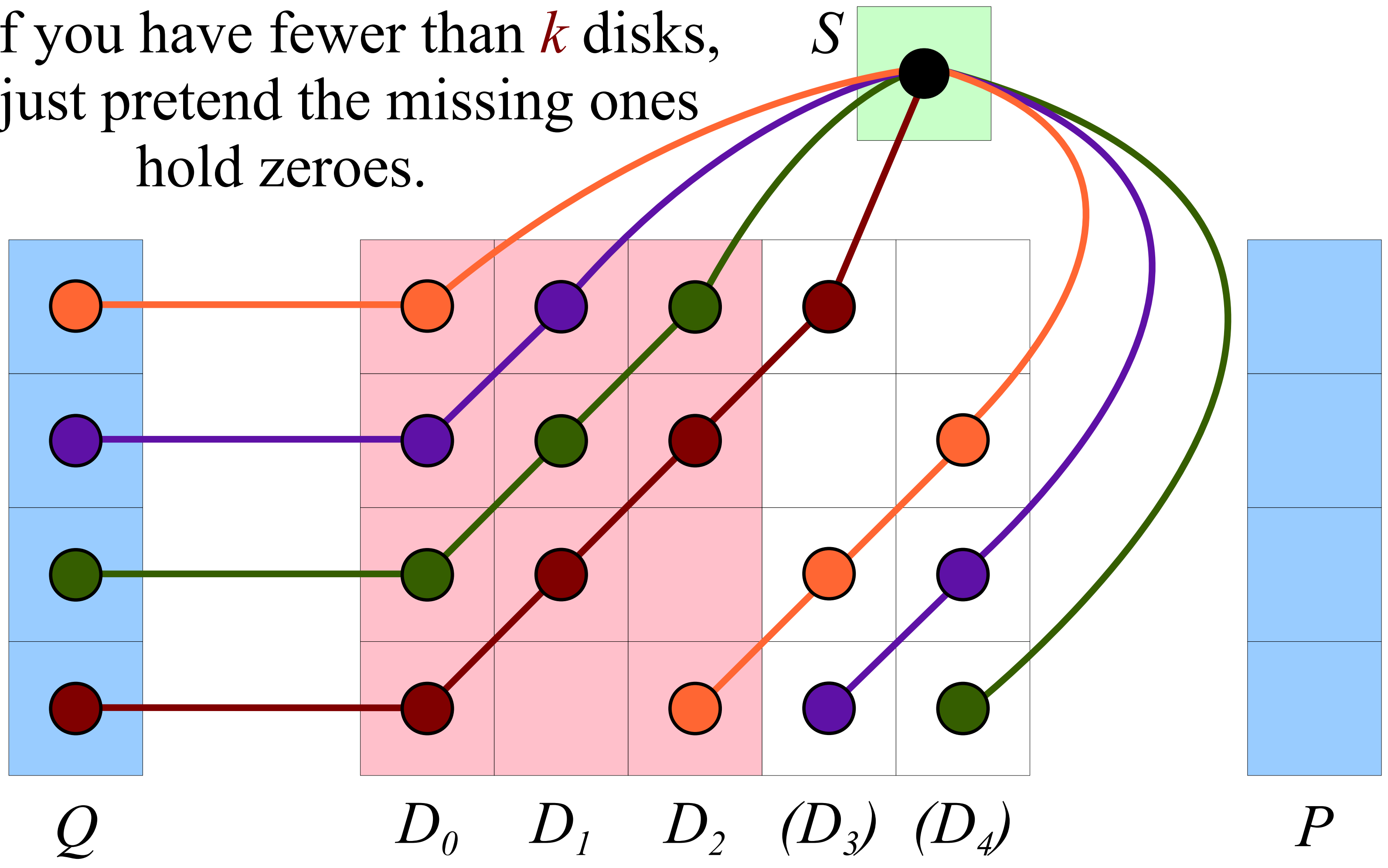


And so on.



Shortening the code.

If you have fewer than *k* disks, just pretend the missing ones hold zeroes.



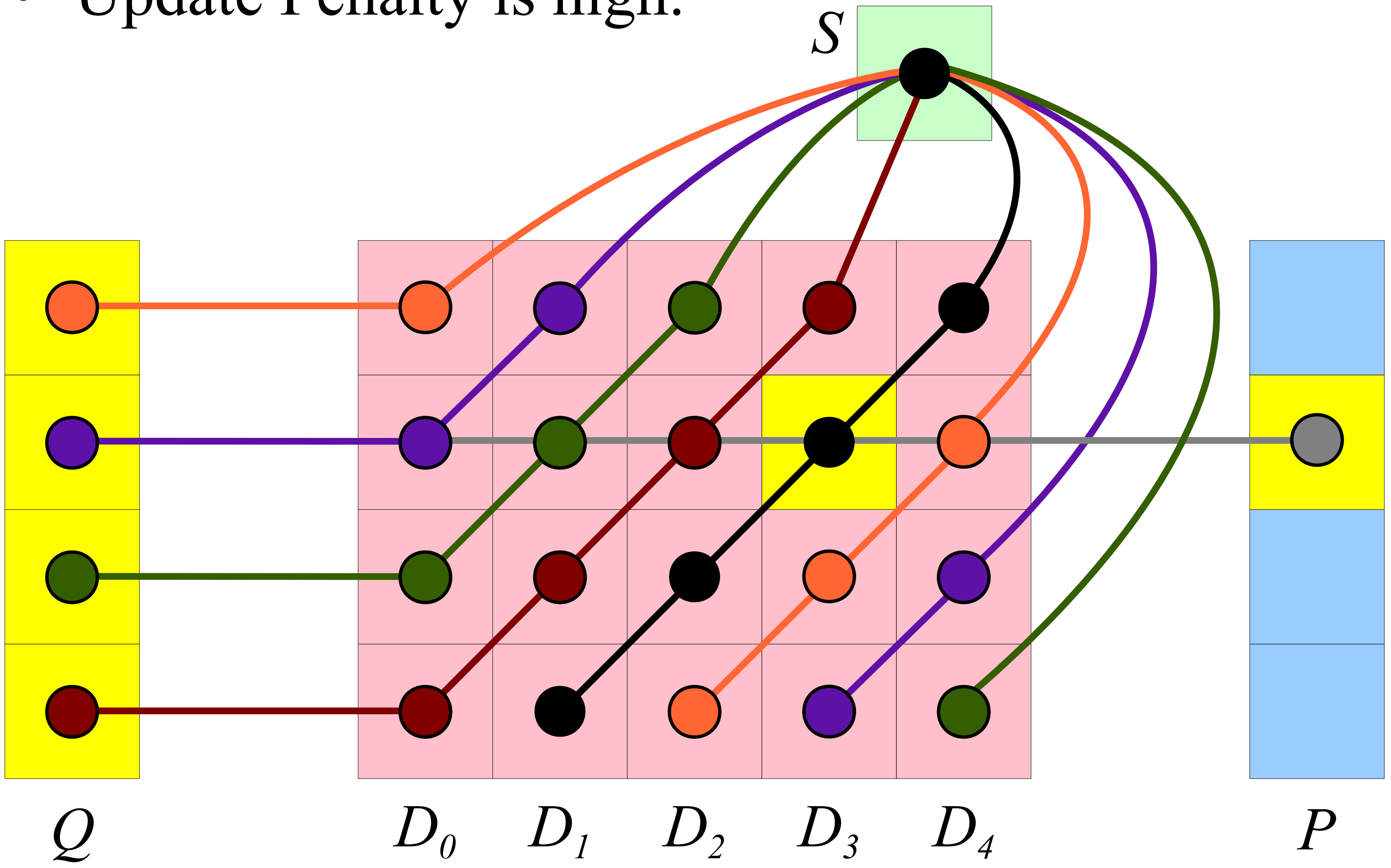
You can do that with any horizontal code.

Benefits of EVENODD

- (Historical): XOR's faster than Galois Field arithmetic.
  - Not as important these days.
- Don't need matrices or linear algebra.
- Recovery from single failures can be faster.
  - (Cheng will tell you about that.)
- Patent no longer enforced (I think).

Downsides of EVENODD

- RDP is faster
- Update Penalty is high.



Code #5: RDP

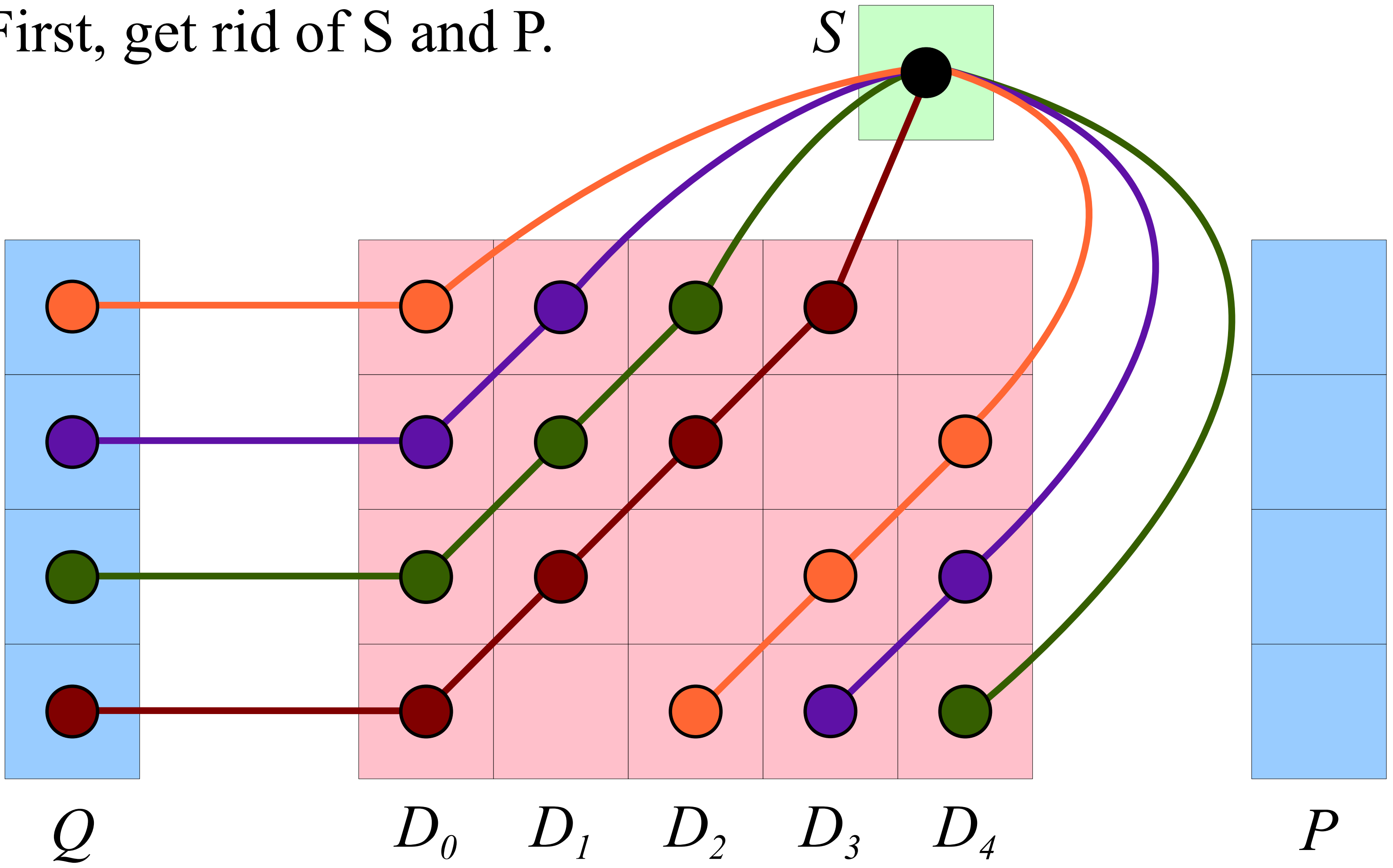
(2004)

RDP – Properties & Constraints

- “Row Diagonal Parity”
- Everything is the same as EVENODD, except you have to subtract one from k.
- So, $k = r$, and $k+1$ must be a prime number.
- Easiest explanation is to convert an instance of EVENODD to an instance of RDP.
  - This is just for the explanation. You don't physically convert EVENODD to RDP.

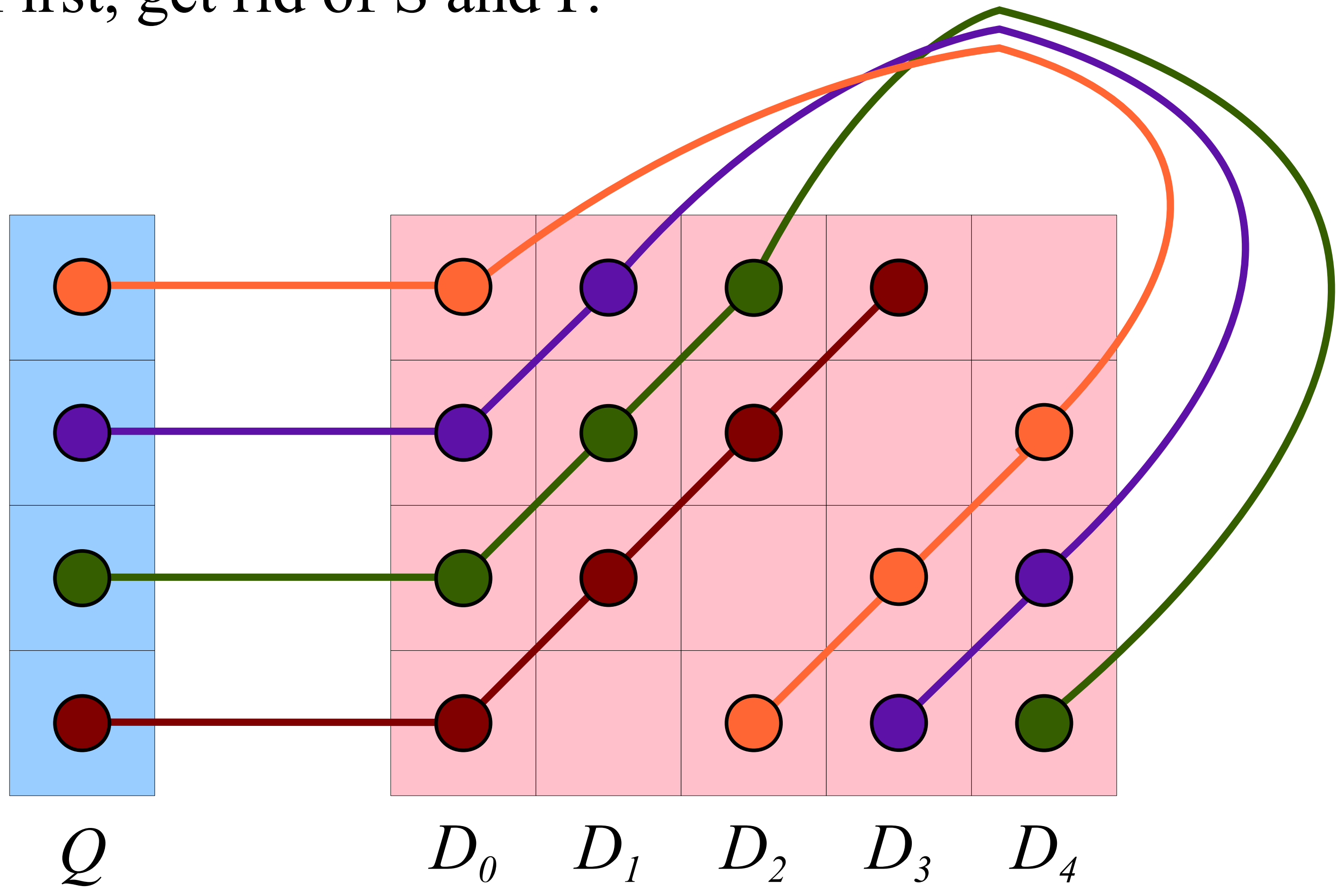
Converting EVENODD to RDP

First, get rid of S and P.



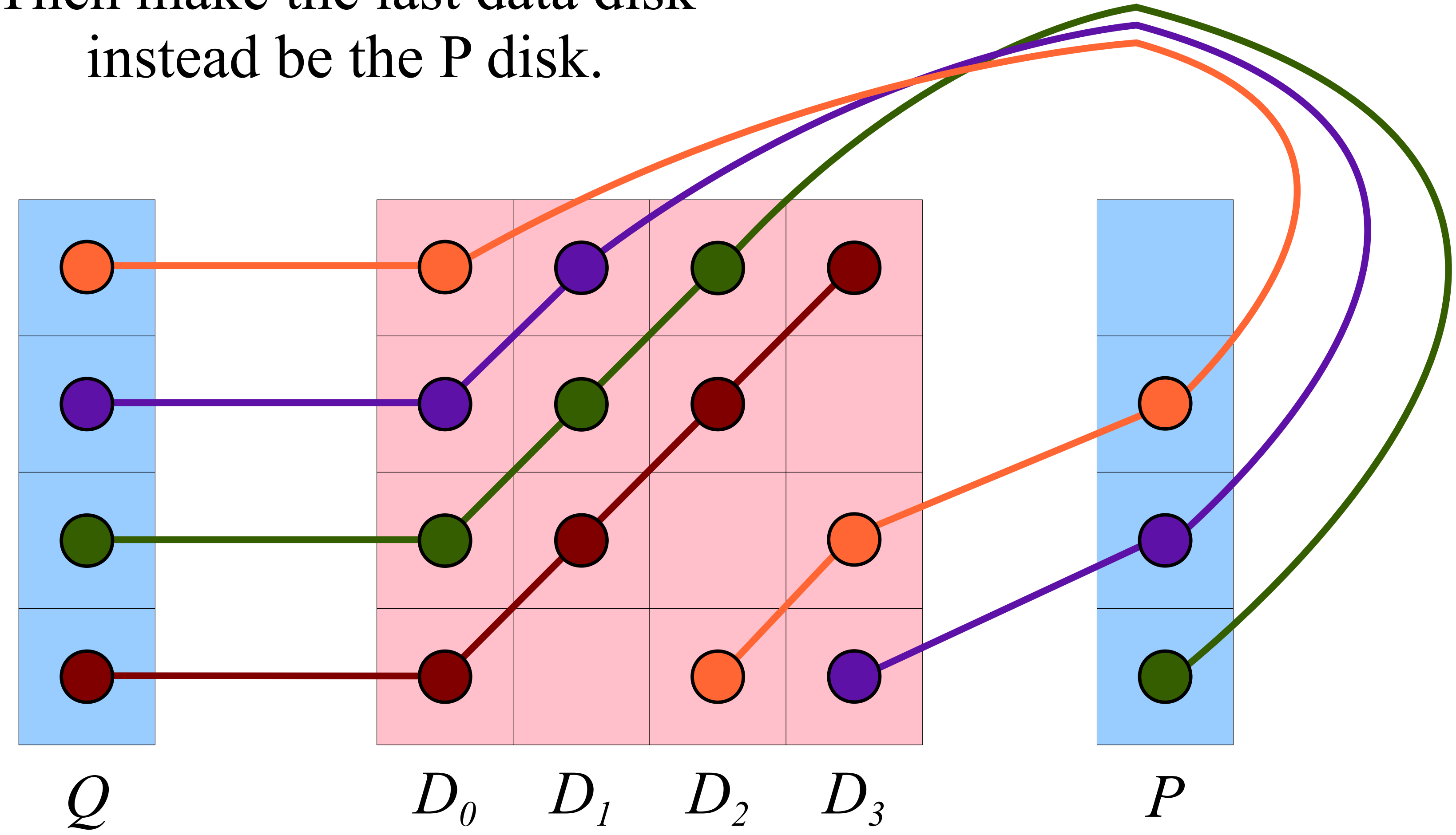
Converting EVENODD to RDP

First, get rid of S and P.



Converting EVENODD to RDP

Then make the last data disk
instead be the P disk.



And you're done!

Benefits of RDP

- Achieves a theoretical minimum on the number of XOR operations required for encoding and decoding.
- Don't have to mess with *S*.

Downsides of RDP

- Has the same update penalty as EVENODD.
- Cannot encode P and Q independently.
- Patent held by Netapp.
  - (unsure of enforcement).

Code #6: X-Code

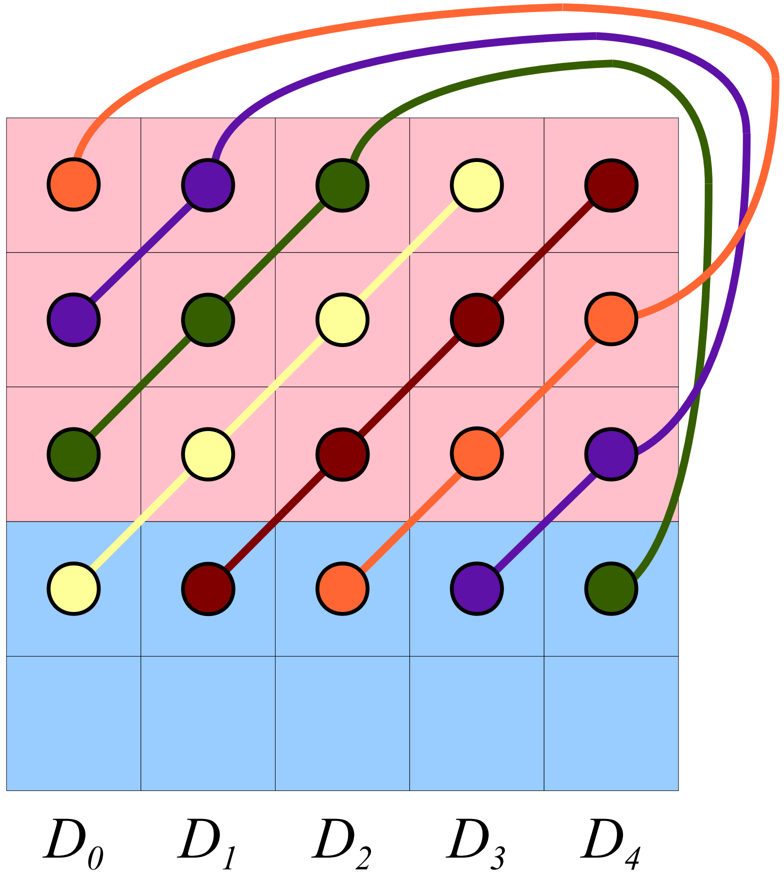
(1999)

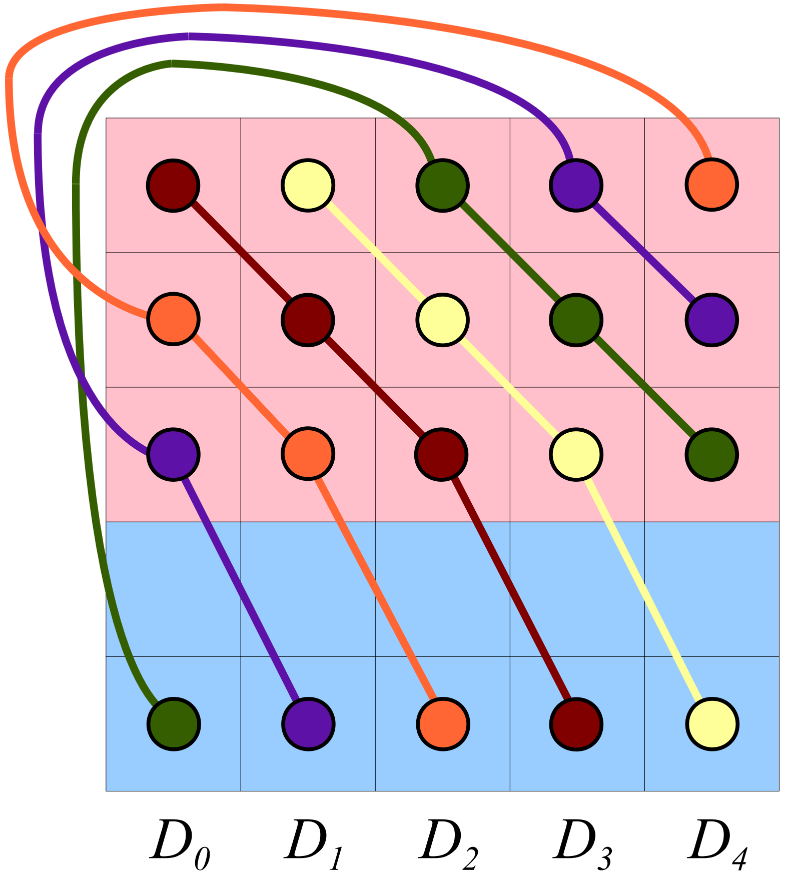
X-Code – Properties & Constraints

- RAID-6 *(m=2)*
- Vertical code
  - Disks must hold data & coding.
- Systematic, MDS
- n must be prime.
- $r = n$.

X-Code Example: $n = 5$ ($k = 3$).

Simple mirror-image diagonal coding.

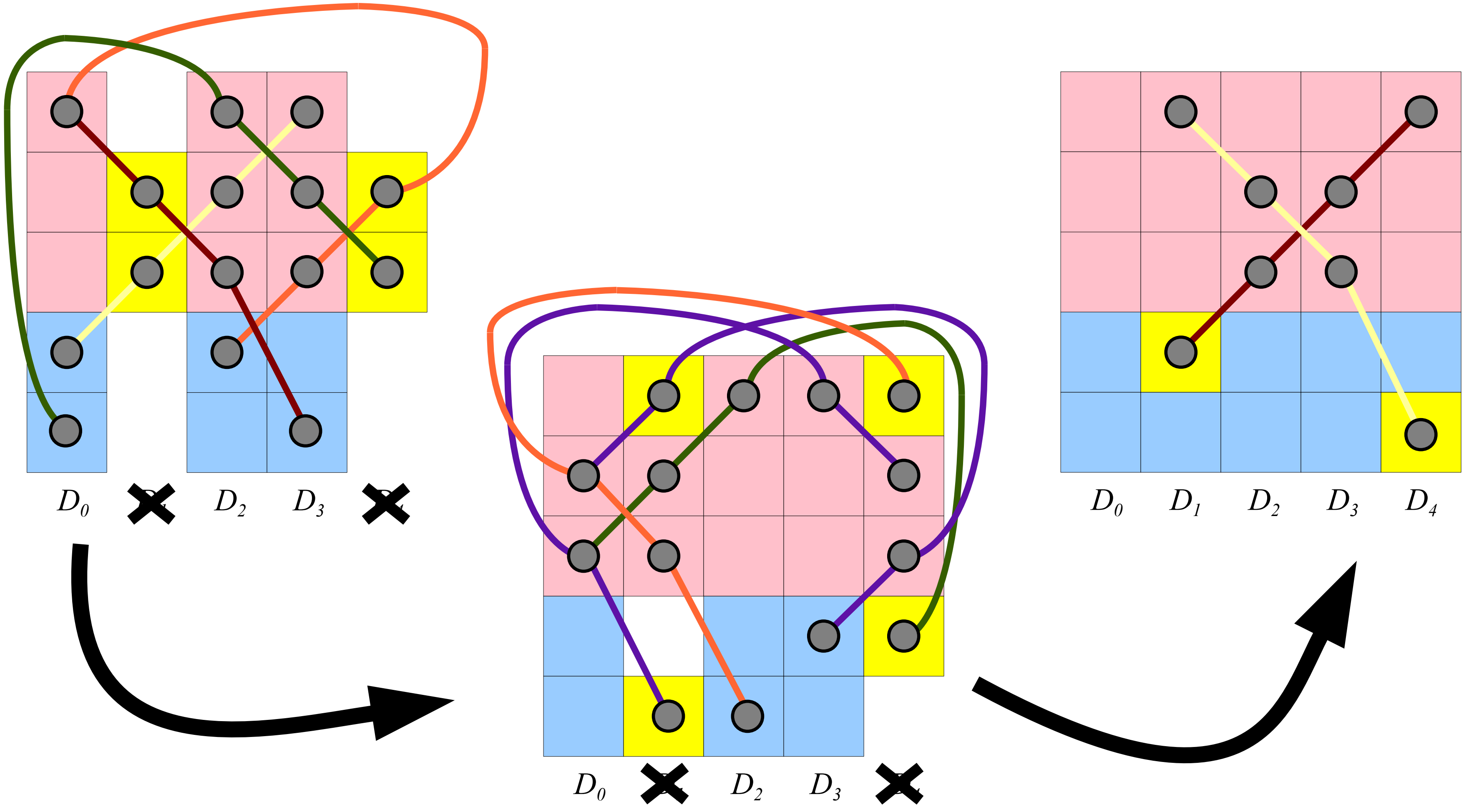




Like EO and RDP, each diagonal chain is missing one disk.

X-Code Example: $n = 5$ ($k = 3$).

Like EVENODD and RDP, find the chains missing one bit and decode from those chains.



Benefits of X-Code

- Achieves a theoretical minimum on encoding, decoding and update penalty.
- Super elegant.

Downsides of X-Code

- Cannot easily shorten a vertical code.
- Patent exists.
- Don't know of any system that uses it.

Codes #7-9: Other Bit-Matrix Codes

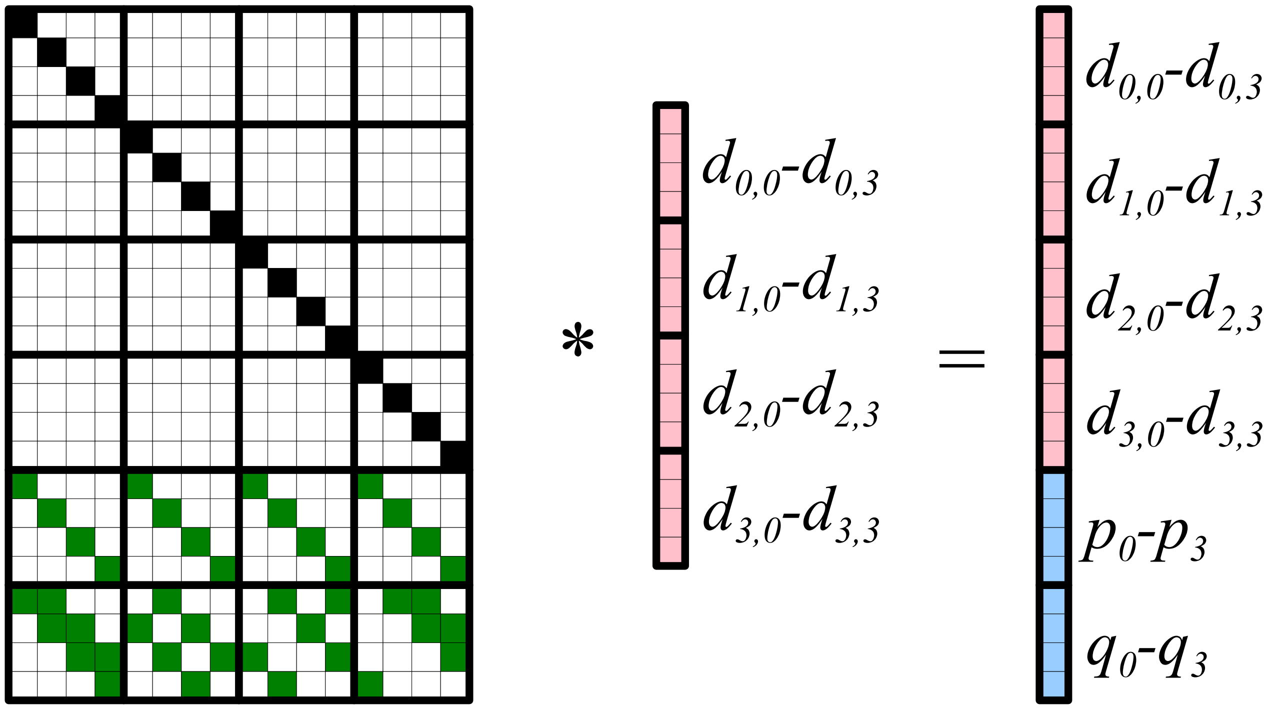
(1999-2009)

Other Bit-Matrix Codes

- Methodology:
  - Use a generator matrix defined by the specific erasure code for k, n, and r. w must equal 1.
  - Only XOR's.
  - Use heuristics to identify common XOR operations to improve performance of both encoding and decoding.
- Benefits:
  - Not really CPU performance (GF ops are fast)
  - Can save you I/O on recovery.

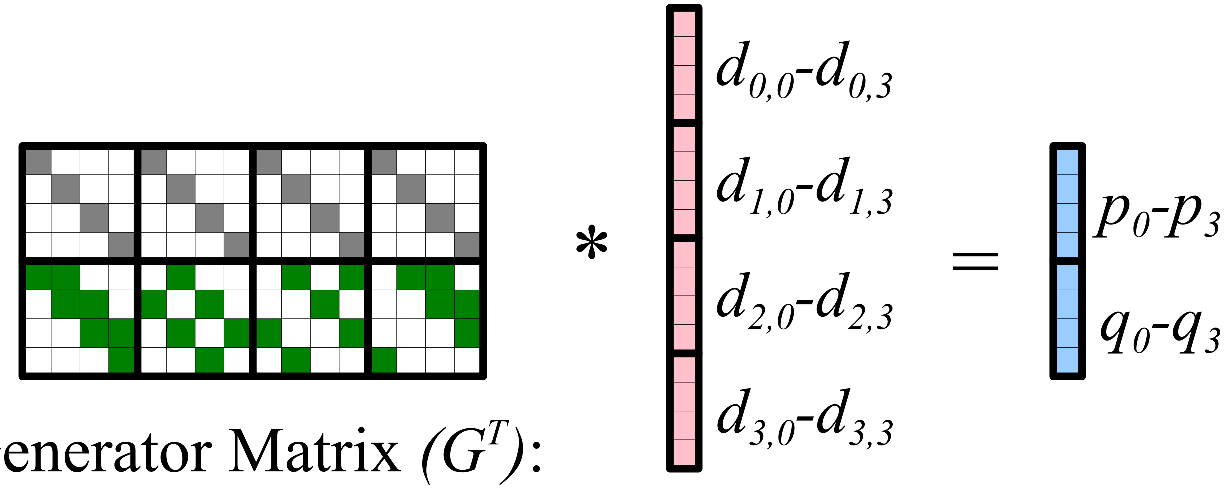
Just to remind you again what it looks like:

Systematic code with *k*=4, *n*=6, *r*=4, *w*=1 (RDP RAID-6)

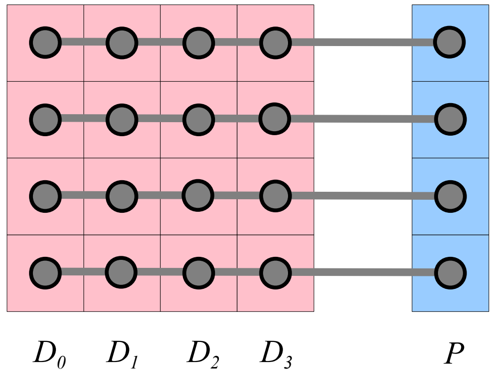


Generator Matrix *(G^T)*:

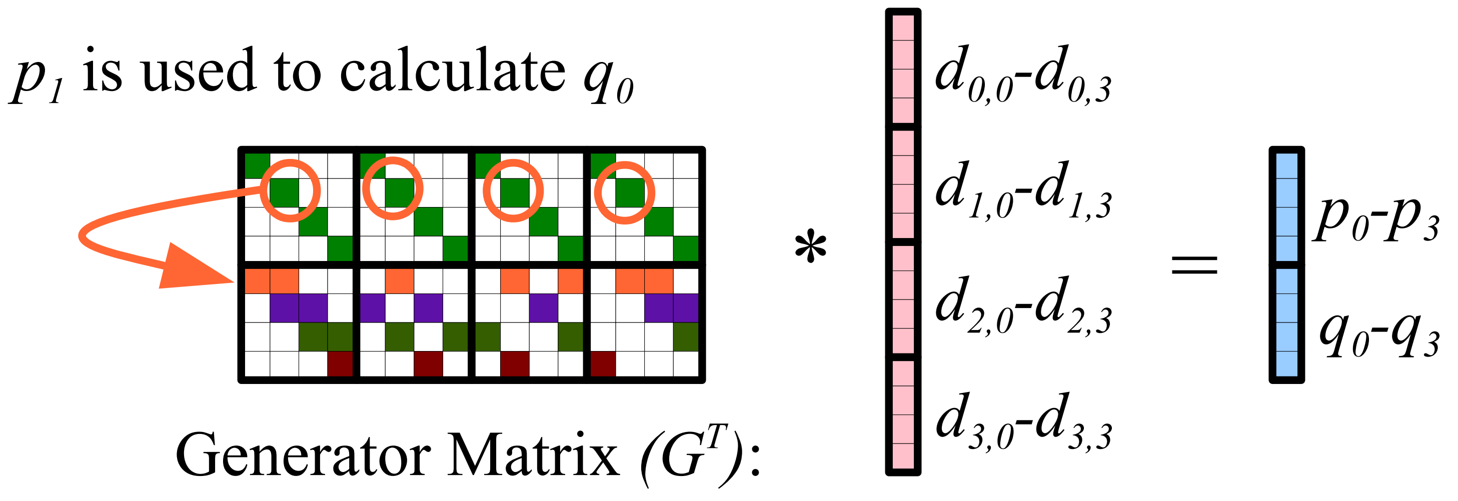
Mapping between Generator, P and Q

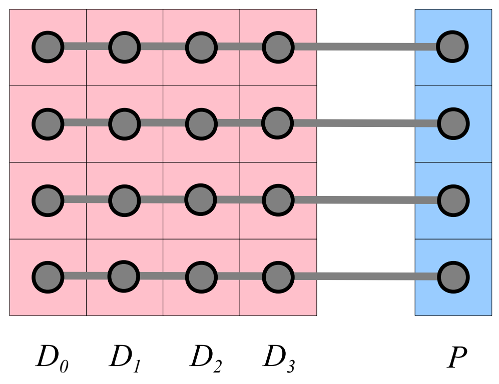


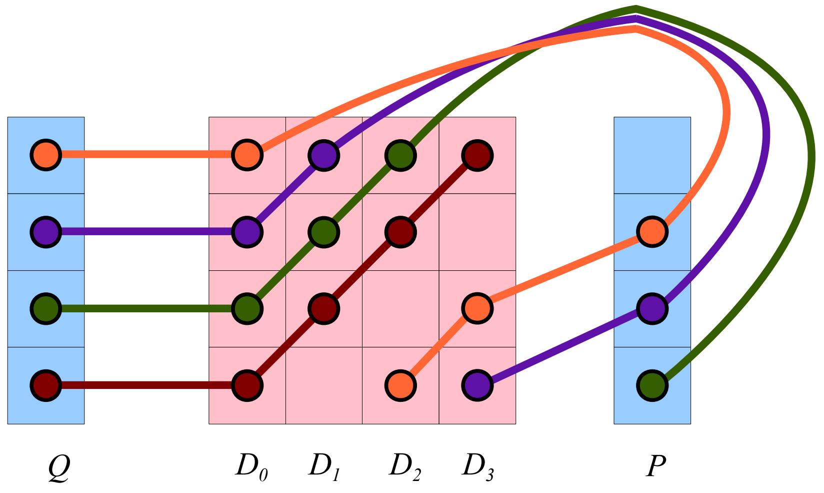
Generator Matrix (G^T):



Mapping between Generator, P and Q





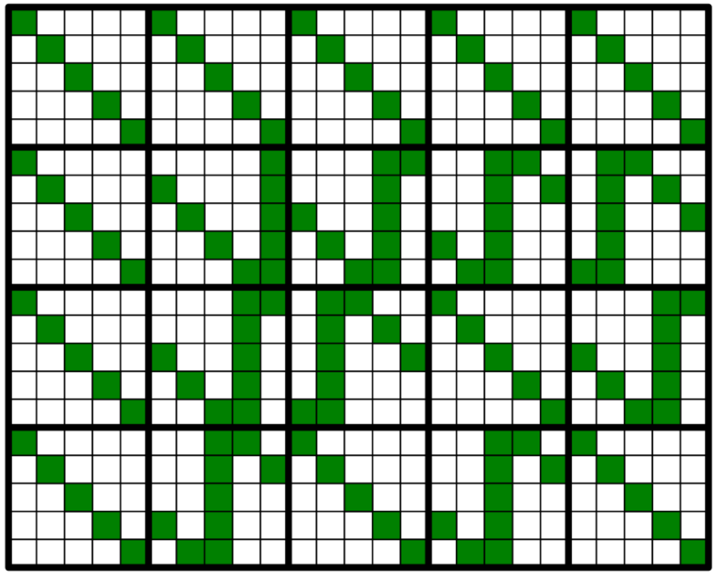


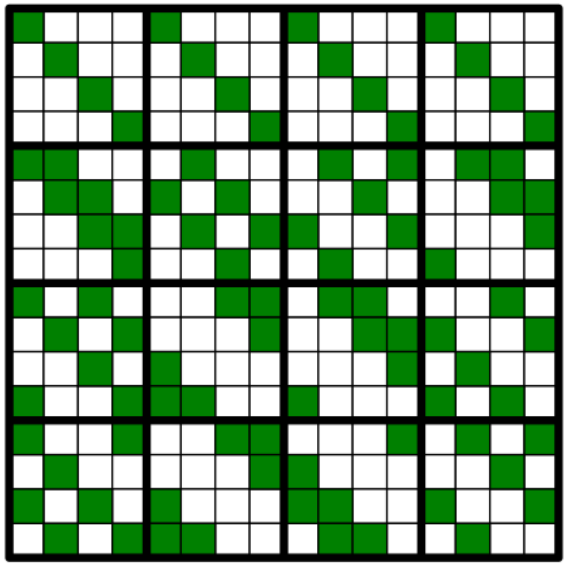
Other Bit-Matrix Codes

- Cauchy Reed-Solomon Codes
  - Discussed previously
- Minimum Density RAID-6 Codes: $k \leq r$
  - Generators have minimum 1's.
  - Better update penalty than EO/RDP.
  - Still good encoding/decoding.
  - Blaum-Roth: $r+1$ prime ($r=4$ and $r=16$: Nice stripes).
  - Liberation: r prime.
  - Liber8tion: $r = 8$ (Nice stripes).

Other Bit-Matrix Codes

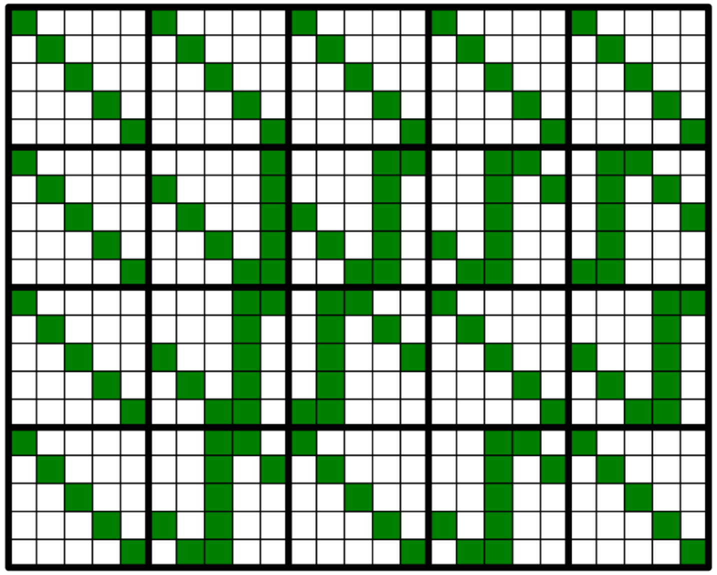
- Generalized EVENODD and RDP:
  - MDS codes for any *k* and *n*, $n \le r$.
  - *r* must be prime and not all primes work (there's a table).
  - Fewer XOR's than Cauchy Reed-Solomon codes.
  - When *m=3*, the generalized EVENODD code is called STAR and has extra documentation on decoding.

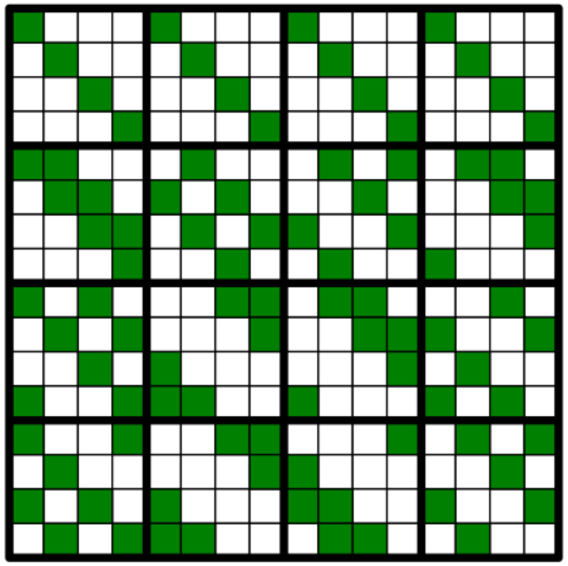




Re-using Intermediates

- Multiple heuristics exist (in bib/open source).
- Identify how to do EVENODD and RDP properly.
- Improve Cauchy encoding and decoding
- Some codes would have unusable decoding performance without the heuristics.





Bit-Matrix Codes: Bottom Line

- XOR Only, Defined with Generator Matrix
- Can have improved recovery over other codes.
- Many Constructions:
  - Cauchy Reed-Solomon (general *k, n*)
  - Blaum-Roth (RAID-6)
  - Liberation, Liber8tion (RAID-6)
  - Generalized EVENODD & RDP (general *k, n*)
- Must use heuristics for reusing intermediates

Open Source Support

Libraries that are not mine:

- Matlab: Has GF arithmetic support.
- Java: Onion Networks FEC library.
  - Slow.
- Luigi Rizzo's Reed-Solomon codec
  - Highly used for years.
  - Easy to read implementation.
- http://www.schifra.com/ (wikipedia link)
  - Gotta pay for documentation
- http://www.ka9q.net/code/fec/

Libraries that are mine:

- GF-Complete: Just released
  - Implemented in C with Kevin Greenan (EMC/Data Domain), Ethan Miller (UCSC) and Will Houston (Tennessee Undergrad / Netapp).
  - All known techniques to facilitate comparison.
  - Single operations *(a*b), a/b, 1/a.*

    w = 1-32, 64, 128.
  - Buffer-constant for w = 4, 8, 16, 32, 64, 128.
  - SSE support for cache line performance

    (see Friday's talk).
  - Submitting implementation paper to TOS.

Libraries that are mine

- Jerasure (version 1.2: 2008)
  - C code written with Scott Simmerman and Katie Schuman.
  - Support for coding and decoding with generator matrices and bitmatrices.
  - Either *$(w > 1 \ \&\& \ r == 1)$* or *$(w==1 \ \&\& \ r > 1)$*.
  - Reed-Solomon support
  - Cauchy Reed-Solomon support
  - Blaum-Roth/Liberation/Liber8tion support
  - Uses old (non-SSE) Galois Field library.

Places that have been using jerasure:

Wings for Media (Startup in Barcelona). University of Hyderabad, India. **Fujitsu Siemens Computers**. BitTorrent. Herzliya Interdisciplinary Center, Israel. Chinese Academy of Sciences. MindTree Ltd (Bangalore, India). University of California, Santa Cruz. Tsinghua University, China. Boston University. **IBM**. Gaikai (Video Gaming Company). Penn State University. Lancaster University (UK). BitMicro Networks. PSG TECH (Coimbatore, India). Dresden University of Technology (Germany). Sultan Qaboos University, Oman. Boston University. Center for Advanced Security Research Darmstadt (Germany). **RSA Laboratories**. Polytechnic University of Turin, Italy. Colorado School of Mines. Humboldt University (Berlin). Fudan University, China. National Institute of Astrophysics, Optics and Electronics (Mexico). New Jersey Institute of Technology. F5 Networks. Archaea Software, Rochester, NY. Quantcast Corporation (San Francisco). Chinese University of Hong Kong. Zhejian University China. Nanyang Technological University, Singapore. LinSysSoft Technologies. University of Waterloo. Cleversafe, Inc. Fern University (Hagen, Germany). Network Appliance, Inc. **HP Labs.** Northwestern University. F5.com (Cloud provider). EPFL, Switzerland. Wayne State University. **Intel Corporation.** AllMyData Inc. Infineon. OnQ Technology, Inc. Hitachi America Ltd. Carnegie Mellon University. Atos Worldline (French IT company). Oregon State University. Villanova University. Ege University International Computer Institute (Turkey). Johns Hopkins University. Cotendo (CDN). Alcatel Bell (Antwerp, Belgium). Zhengzhou University China. Oracle. Shantou University, China. Jiaotong University, China. Aalborg University. University of Manchester (England). Daegu University, South Korea. University of Ottawa. Connected Lyfe. **Microsoft**.

Libraries that are mine

- <u>Jerasure 2.0 planned for August</u>
  - Will use the new GF library
  - Support for Generalized EO and RDP
  - Parity Check Matrix decoding
  - General *k, n, w, r*
  - Horizontal, Vertical, Non-systematic codes
  - Support for non-MDS and regenerating codes
  - More support for matrix manipluation & checking
  - Multicore support
  - Maybe a C++ interface. Maybe not.

Libraries that are mine

- <u>"Uber-CSHR" and "X-Sets"</u>
  - Programs for reuse of intermediate sums in bit-matrix code.
  - Implements all published techniques to solve this problem (Hafner, Huang, Plank).
  - Will be integrated with the next jerasure.

Time to take a break!

(Remember, the notes include an annotated bibliography, ordered by the topics in the slides, so that you can find more reference material for all of this.)