UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Oliver Pikani

Comparative Analysis of Erasure Correcting Codes in Data Storage

Bachelor's Thesis (9 ECTS)

Supervisor:
Irina Bocharova, PhD

Tartu 2025

Comparative analysis of erasure correcting codes in data storage

Abstract:

Erasure-correcting codes add redundancy to data, allowing the recovery of original data in case of erasures, thereby reducing the risk of data loss. This thesis analyzes the application of different erasure-correcting codes in storage systems. Reed-Solomon codes offer the maximum possible erasure correction for a given amount of redundancy, but they are computationally expensive, while XOR-based array codes are computationally efficient, making them ideal for RAID systems. LDPC codes are well-suited for large distributed storage systems, while convolutional codes show promise in simpler storage environment. The thesis also includes the implementation and comparison of a convolutional encoder and two decoding algorithms.

Keywords: Erasure correcting codes, Storage system, RAID, Convolutional codes

CERCS: P170 Computer science, numerical analysis, systems, control

Veaparanduskoodide võrdlev analüüs salvestussüsteemides

Lühikokkuvõte:

Veaparanduskoodid lisavad andmetele liiasust, võimaldades taastada algandmeid rikete korral ning seeläbi vähendades andmekao riski. Töös analüüsitakse erinevate veaparanduskoodide rakendusi salvestussüsteemides. Reed-Solomoni koodid pakuvad maksimaalset võimalikku veaparandusvõimekust antud salvestusmahu juures, kuid eeldavad suurt arvutusressurssi. Seevastu XOR-põhised array-koodid on arvutuslikult soodsad ja sobivad hästi RAID-süsteemidesse. LDPC-koodid sobivad kasutamiseks suurtes salvestussüsteemides, samas kui konvolutsioonilised koodid on perspektiivikad lihtsamates salvestussüsteemides. Töö sisaldab ka konvolutsioonilise kodeerija ning kahe dekodeerija implementeerise protsessi kirjeldust ning nende võrdlust.

Võtmesõnad: Veaparanduskoodid, Salvestussüsteemid, RAID, Konvolutsioonilised koodid

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1. Introduction

Many different erasure-correcting codes have been developed to improve the reliability of data in storage systems. These codes offer different trade-offs between storage efficiency, computational complexity, and the ability to recover from disk failures. However, because there are so many types of codes and decoding methods, it is not always clear which one is the most suitable for real-world storage systems. The choice often depends on the specific requirements of the system, and there is no single code that fits all situations.

This thesis looks at that problem by comparing four different classes of erasure-correcting codes: Reed-Solomon codes, parity-array codes (with a focus on EVENODD), Low-Density Parity-Check (LDPC) codes, and convolutional codes. These codes follow different approaches and have different strengths and weaknesses. Rather than identifying one “best” code, the goal is to evaluate which types of codes perform better in different types of storage scenarios, and under what conditions.

Additionally, this thesis focuses on the practical use of convolutional codes for storage, which are not commonly deployed in real systems. Two convolutional decoders are implemented: a sliding-window decoder that solves a linear system at each step, and a peeling decoder that uses an iterative process based on parity checks. These implementations are tested and compared based on decoding performance in the presence of erasures. The results help assess whether convolutional codes, with their continuous encoding and simple structure, could be a suitable choice for low-complexity storage systems.

The thesis is organized as follows: it starts with background concepts and an overview of how erasure codes are used in storage. It then introduces the four code families, focusing on how they are structured and decoded. Finally, the convolutional decoders are implemented, tested, and compared. The work concludes with a summary of the findings based on these results.

By examining how different codes behave in practice, this thesis aims to offer insights into which coding approaches are more effective or practical in various storage settings.

2. Background

Storage systems have become so large and complex that failures are now considered unavoidable. To prevent data loss, system designers should proactively develop strategies to ensure data remains secure and recoverable when failures happen [1].

J. S. Plank [2] explains that in large storage systems, protecting data against loss due to failures is essential. Erasure codes are key to this protection because they help systems recover lost data when components fail. However, he notes that erasure codes can introduce considerable performance costs during two key processes: encoding, where redundancy is calculated for new data, and decoding, where lost data is reconstructed after a failure.

Erasure coding is increasingly replacing traditional replication in storage systems. Unlike replication, which simply creates copies of data, erasure coding is a more efficient method of data protection. This shift has introduced new challenges in system design and research, as mentioned by R. K. Vinayak [3].

Disk sectors often include extra error-correcting data on its own to handle small errors like when a few bits are flipped. However, when more significant errors occur, such as multiple bit flips or hardware failures, the system treats it as an erasure, losing the data on the affected disk. To address such issues, storage systems use erasure codes to add redundancy and tolerate failures [1].

2.1 Preliminaries

Before going deep into erasure codes, it's important to understand a few fundamental concepts.

A *code* is typically denoted as (n, M, d), where n is the length of each codeword, M is the number of codewords in the code, and d is the minimum distance between any two codewords [4].

Let GF(q) denote a *finite field* with q elements, where $q = p^m$, with p being a prime number and m a positive integer. For example, GF(2) consists of the elements $\{0,1\}$, and arithmetic operations are performed modulo 2 [4].

Definition 1. The *Hamming distance* between two vectors $\boldsymbol{x} = (x_1,\ x_2, \dots, x_n)$, $\boldsymbol{y} = (y_1,\ y_2, \dots, y_n)$, where $x_i, y_i \in \mathrm{GF}(q)$ is the number of coordinates on which $\boldsymbol{x}$ and $\boldsymbol{y}$ differ. This is denoted as $d(\boldsymbol{x}, \boldsymbol{y})$ [4].

Definition 2. Let C be an (n, M, d) code over $\mathrm{GF}(q)$ with $M > 1$. The *minimum distance* of C, denoted by d, is the smallest Hamming distance between any two distinct codewords in C [4].

Definition 3. Let C be an (n, M, d) code over a finite field $\mathrm{GF}(q)$. The $[n, k]$ code C over $\mathrm{GF}(q)$ is called *linear* if it forms a k-dimensional linear subspace of $\mathrm{GF}(q)^n$. That is, for any two codewords $\boldsymbol{c_1}, \boldsymbol{c_2} \in C$ and any scalars $\mathrm{a}_1, \mathrm{a}_2 \in \mathrm{GF}(q)$, the linear combination $\mathrm{a}_1\boldsymbol{c_1} + \mathrm{a}_2\boldsymbol{c_2}$ is also in C [4].

Definition 4. A *generator matrix* of a linear $[n, k]$ code over $\mathrm{GF}(q)$ is a $k \times n$ matrix whose rows form a basis of a linear subspace. We typically denote a generator matrix by G. The rank of G is equal to the dimension of the code C [4].

Definition 5. Let C be a linear $[n, k]$ code over $\mathrm{GF}(q)$. A *parity-check matrix* of C is an $r \times n$ matrix H, where $r = n - k$ over $\mathrm{GF}(q)$ such that a vector $\mathbf{c} \in \mathrm{GF}(q)^n$ is a codeword in C if and only if it satisfies $H\boldsymbol{c}^T = 0$ [4].

This means that every valid codeword must follow a set of predefined rules represented by H. Multiplying H by a vector $\boldsymbol{c}$ checks whether the codeword is correct — if the result is a zero vector, the vector $\boldsymbol{c}$ is a valid codeword; if not, errors have occurred.

Definition 6. *XOR(x, y)* is a bitwise operation, that outputs 1 when the two bits are different, meaning it results in 1 if one bit is 1 and the other is 0. This behavior is equivalent to addition modulo 2 [5].

Definition 7. *MDS codes (Maximum Distance Separable codes)* are error-correcting codes that provide the highest possible error detection and correction capability while using the minimum amount of redundancy. These codes encode k data symbols into n symbols by adding $n - k$ redundant symbols and ensure that the minimum distance between different codewords $d = n - k + 1$ is maximized. This property allows MDS codes to correct up to $\left\lfloor \frac{n-k+1}{2} \right\rfloor$ errors or recover data in the event of up to $n - k$ erasures [6].

MDS codes are commonly used in data storage systems where efficient error correction and data recovery are critical.

Definition 8. Given an $[n, k, d]$ code C over GF(q), let $\boldsymbol{c} \in C$ be the transmitted codeword and $\boldsymbol{y} \in GF(q)^n$ be the received word. An *error* occurs when one or more entries in $\boldsymbol{c}$ are altered during transmission or storage. The number of errors is given by the Hamming distance $d(\boldsymbol{y}, \boldsymbol{c})$, which counts the number of positions where $\boldsymbol{c}$ and $\boldsymbol{y}$ differ. The *error locations* correspond to the indices of these differing positions [4].

Definition 9. An *erasure* occurs when the position of a missing codeword entry is known, but its value is unknown [4].

Definition 10. *Rate* R of a (n, M) code is $\frac{\log_2 M}{n}$. For $[n, k]$ linear code, R is equal to $\frac{k}{n}$. A higher code rate means less redundancy and better storage efficiency but reduces fault tolerance [7].

3. Basics of Erasure Coding in Storage

In modern storage systems, data is protected against disk failures using erasure codes, which introduce redundancy in the form of additional symbols stored across multiple disks. When a disk fails, it becomes unavailable (usually shuts down), and the system is able to detect which specific disk has failed. This type of failure is known as an erasure, since the location of the missing data is known. This is in contrast to an error, where a disk might return incorrect data without any indication [8].

Typically, a storage system consists of n disks, with k used for storing data and $r = n - k$ for redundancy. The data stored on the k data disks is encoded to generate r additional redundant symbols. These are stored on r redundant disks, allowing the system to tolerate up to r disk failures. When failures occur, the original data can be recovered by decoding the information from the remaining disks [1].

Each redundant disk C_i stores information computed by a function F_i, which combines the content of all k data disks. The function F is defined by the coding method in use. In other words, the code specifies how redundancy is calculated. For example, in storage systems that use Reed-Solomon (RS) codes, each disk is divided into fixed-size words, and the encoding operations are performed on these word units. Each word consists of m bits and is treated as a symbol in the finite field $GF(2^m)$, where addition and multiplication follow the rules of that field [8].

A common method for constructing erasure codes is to use arithmetic over a finite field, typically $GF(2^m)$. In this setting, each symbol, whether a data symbol or a redundant, is treated as an element of the field. Encoding is typically performed by computing linear combinations of the data symbols using coefficients drawn from the same field, often represented as a matrix multiplication between a data vector and a generator matrix. To verify correctness or recover missing symbols, decoding involves solving systems of linear equations over $GF(2^m)$, often using a parity-check matrix [4].

4. Classes of erasure-correcting codes

Various erasure-correcting codes are employed in modern storage systems to ensure data reliability and fault tolerance. This section focuses on four classes of such codes: Reed-Solomon, Parity-Array, LDPC and Convolutional codes.

4.1 Reed-Solomon codes

Named after two of their inventors, Reed-Solomon (RS) codes are among the most widely used error-correcting codes due to several advantages. They are MDS codes, which means that the correction of the largest possible number of erasures or errors for a given code length is guaranteed. For any RS(n, k) code, the minimum distance is $d = n - k + 1$. As linear block codes, RS codes support efficient encoding and decoding, and their algebraic structure allows implementation using relatively simple hardware [4].

An RS code over a field GF(2^m) is typically denoted as RS$(n,\ k)$, where:

- n is the total number of symbols in a codeword,
- k is the number of data symbols,
- r = $n - k$ is the number of redundant symbols, and
- each symbol is an element of GF(2^m), i.e., a m-bit word.

These parameters define the basic structure of RS codes [9].

4.1.1 Field elements

The following explanation of RS codes is adapted from [7]:

The field GF(2^m) is constructed using binary polynomials of degree less than m, with coefficients in GF(2). Arithmetic operations, such as addition and multiplication, are performed modulo an irreducible polynomial of degree m, which defines how multiplication is reduced back into the field. An irreducible polynomial is one that cannot be factored into lower-degree polynomials over the same field, similarly to a prime number in integer arithmetic.

If the irreducible polynomial is also a primitive polynomial, which we can choose and prefer, then its root a is a primitive element of the field. This means that every nonzero element in GF(2^m) can be written as a power of a:

$$\{\alpha^0, \alpha^1, \ldots, \alpha^{2^m-2}\}.$$

The value $2^m - 1$ comes from the fact that the field GF(2^m) contains exactly $2^m - 1$ nonzero elements. These elements form a cyclic multiplicative group of order $2^m - 1$, where a primitive element a generates all nonzero values through its powers.

For example, GF(8) (which is GF(2^3)) can be constructed using the primitive polynomial:

$$p(x) = x^3 + x + 1.$$

Here, α is a root of $p(x)$ and a primitive element, meaning:

$$\alpha^7 = \alpha^0 = 1.$$

The nonzero elements of GF(8) are then represented as:

$$\{1\ , \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6\}.$$

This allows multiplication to be performed by adding exponents modulo 7.

Table 1. Multiplicative group of GF (2^3)

Power of primitive element	Polynomial form	Binary form
α^0	1	001
α^1	α	010
α^2	α^2	100
α^3	$\alpha^3 = 1 + \alpha$	011
α^4	$\alpha + \alpha^2$	110
α^5	$\alpha^2 + \alpha^3 = 1 + \alpha + \alpha^2$	111
α^6	$\alpha + \alpha^2 + \alpha^3 = 1 + \alpha^2$	101
$\alpha^7 = \alpha^0$	$\alpha + \alpha^3 = 1$	001

The table illustrates the multiplicative group of GF(2^3), showing how each power of the primitive element α corresponds to a unique field element in both polynomial and 3-bit binary vector forms.

4.1.2 Generator Polynomial and Code Construction

In RS codes, each valid codeword is a multiple of a generator polynomial $g(x)$, which defines the structure of the code. This polynomial is the lowest-degree polynomial that divides $x^n - 1$, ensuring that all codewords $c(x)$ satisfy:

$$c(x) \equiv 0 \; mod \; g(x).$$

This means every codeword polynomial leaves no remainder when divided by $g(x)$, and can be written as:

$$c(x) = m(x) \cdot g(x).$$

where $m(x)$ is the message polynomial of degree less than k.

The degree of $g(x)$ is $n - k$, which implies the code has dimension k. The set:

$$\{ g(x), x^1 g(x), x^2 g(x), \dots, x^{k-1} g(x) \}$$

forms a basis for the code and defines the generator matrix G, which can be used for matrix-based encoding.

To guarantee the minimum distance $d = n - k + 1$, the generator polynomial must include $d - 1 = n - k$ consecutive powers of a primitive element α in GF(2^m) as its roots. The starting exponent b is a design parameter, often set to 1. The generator polynomial is then constructed as:

$$g(x) = (x - \alpha^b)(x - \alpha^{b+1}) \cdots (x - \alpha^{b+n-k}).$$

Example:

Consider an RS(7, 3) code over GF(8), where the field is constructed using the irreducible polynomial $p(x) = x^3 + x + 1$, and α is a primitive element. If the generator polynomial is constructed using roots $\alpha^1, \alpha^2, \alpha^3, \alpha^4$, we have:

$$\begin{aligned} g(x) &= (x - \alpha^1)(x - \alpha^2)(x - \alpha^3)(x - \alpha^4) \\ &= x^4 + \alpha^{13} x^3 + \alpha^6 x^2 + \alpha^3 x + \alpha^{10}. \end{aligned}$$

This polynomial has degree 4, consistent with $n - k = 4$, and ensures that all codewords are divisible by $g(x)$.

4.1.3 Generator matrix

The generator matrix G of an RS(n, k) code over GF(2^m) is constructed using a Vandermonde matrix. This matrix is built by evaluating message polynomials at distinct field elements.

Let α be a primitive element of GF(2^m), such that every non-zero element in the field can be expressed as a power of α. The generator matrix G is formed by selecting k distinct non-zero elements $\beta_1, \beta_2, \dots, \beta_k$ from GF(2^m), where each β_i is a power of α, i.e., $\beta_i = \alpha^{e_i}$ for distinct exponents e_i. The generator matrix G is then given by the $k \times n$ Vandermonde matrix:

$$G = \begin{bmatrix} 1 & \beta_1 & {\beta_1}^2 & \cdots & {\beta_1}^{n-1} \\ 1 & {\beta_2}^1 & {\beta_2}^2 & \cdots & {\beta_2}^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \beta_k & {\beta_k}^2 & \cdots & {\beta_k}^{n-1} \end{bmatrix}.$$

Each row shows the results of plugging a different field value (β_i) into the polynomials $1, x, x^2$, etc. The columns spread these results across the entire codeword. As an example, for the RS(7, 3) code, the generator matrix is:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \alpha^5 & (\alpha^5)^2 & (\alpha^5)^3 & (\alpha^5)^4 & (\alpha^5)^5 & (\alpha^5)^6 \\ 1 & \alpha^6 & (\alpha^6)^2 & (\alpha^6)^3 & (\alpha^6)^4 & (\alpha^6)^5 & (\alpha^6)^6 \end{bmatrix},$$

where all exponents are computed modulo 7, since $\alpha^7 = 1$. This matrix is of size 3×7 and its rows are constructed from distinct powers of α, ensuring the linear independence required for the MDS property.

Each message vector $\boldsymbol{m} = [m_0, m_1, m_2]$ is encoded to a codeword $\boldsymbol{c} = \boldsymbol{m} \cdot G$. This matrix-based method gives the same result as encoding with the generator polynomial $g(x)$ and both produce valid codewords.

4.1.4 Encoding and Decoding

Encoding can be done either by polynomial multiplication, where $c(x) = m(x) \cdot g(x)$, or by matrix multiplication, where $\boldsymbol{c} = \boldsymbol{m} \cdot G$.

Decoding involves computing syndromes from the received vector and solving a system of linear equations to locate and correct errors. RS codes can correct up to $n - k$ erasures.

4.1.5 Performance

Reed-Solomon codes are MDS codes capable of correcting up to $r = n - k$ erasures, making them very reliable for storage applications. Unlike many other codes, RS codes can be constructed for any valid combination of n and k. However, this strength comes with computational cost. Encoding has a complexity of $O(n^2)$. Addition (XOR) is cheap and fast, but multiplication over GF(2^m) can be expensive. Systematic encoding simplifies RS encoding by placing the original message directly in the codeword and computing only the parity part. This avoids altering the message and makes decoding slightly less complex. For small m (e.g., ≤ 8), finite field multiplications are accelerated using precomputed lookup tables. For larger m, full multiplication tables become too large to store, so log/antilog tables or iterative multiplication methods are needed, which can reduce performance.

Decoding complexity is $O(rn)$ on average or $O(n^3)$ in the worst case when solving a large system of linear equations during reconstruction. For large systems, decoding performance is often bottlenecked by the number of devices involved in reconstruction.

Updates to a single data symbol require recomputing all r redundant symbols, resulting in $O(r)$ complexity per update, with $r + 1$ XORs and up to r multiplications per symbol.

In practice, RS codes are very well-suited for systems that prioritize data durability, but their higher processing cost makes them less suitable for systems with low latency or limited resources, unless hardware acceleration is used [10].

4.2 Parity-array codes

Array codes for storage systems emerged in the 1990s, driven by the need to eliminate Galois Field arithmetic and implement error correction using only XOR operations. Not all array codes achieve maximum fault tolerance for their redundancy. The parity code discussed next, EVENODD, is a MDS variant. EVENODD, as introduced in [4], is a 2-erasure correcting code that achieves optimal redundancy for the case of r = 2 redundant disks. This means it can tolerate any two disk failures without data loss. Its reliance solely on XOR operations for

encoding and decoding makes it particularly suitable for systems that require fast recovery from disk failures with minimal overhead [11].

In an array code, each disk has multiple sectors, forming a disks × sectors-per-disk array. In the case of EVENODD, if the code uses k data disks, the array consists of $k + 2$ columns (disks) and $k - 1$ rows (sectors), forming a $(k - 1) \times (k + 2)$ array. The first k columns hold the data, while the two redundant columns store horizontal and diagonal parity, respectively. Each redundant sector is computed as a linear combination of the data sectors, enabling efficient recovery by solving linear equations within the array [1].

The EVENODD code uses an extra "imaginary row" during encoding. This row is not stored and contains only zeros. It is used only in the calculation of the diagonal parity. With this row, the array has a total of k rows, even though only $k - 1$ real rows are stored on disk [11].

Using exactly $k - 1$ real rows is important because it keeps the parity equations linearly independent. The diagonal parity is based on modulo k indexing, which means it wraps around every k steps. This only works correctly if the array has k rows, including the imaginary one. The imaginary row ensures that each diagonal intersects every column exactly once. This helps the code stay MDS. Without the imaginary row, the code would not work correctly for all failure cases [11].

The role of the imaginary row is illustrated in the following example.

In practice, each disk has far more than $k - 1$ sectors (e.g., billions of sectors in multi-terabyte drives). The EVENODD encoding is therefore applied repeatedly across the disk: each group of $k - 1$ consecutive sectors per disk is treated as a stripe unit, and each stripe is protected independently using the $(k - 1) \times (k + 2)$ array structure [11].

To prevent bottlenecks caused by repeated write operations, redundancy can also be distributed across all disks rather than dedicated to specific ones. This structure serves as an extension of RAID-4 (which uses dedicated parity) but can also be adapted to RAID-5 (which distributes parity) [11].

The encoding of each stripe can also be described using a binary generator matrix G of size $bk \times bn$, where $b = k - 1$ is the number of rows per stripe, k is the number of data columns, and $n = k + 2$ is the total number of columns. The matrix has the form:

$$G = [\, I_{bk} | \, P \,].$$

Here, I_{bk} is the identity matrix of size $bk \times bk$, and P is the parity matrix of size $bk \times b(n-k)$ that encodes the horizontal and diagonal parity. The generator matrix encodes each data stripe of bk bits into a codeword of bn bits [10].

For example, in a [7,5] EVENODD code and block size $b = 4$, the generator matrix G is of size 20×28, and can be composed of block matrices as follows:

$$G = \begin{bmatrix} I_4 & 0 & 0 & 0 & 0 & P0_0 & P1_0 \\ 0 & I_4 & 0 & 0 & 0 & P0_1 & P1_1 \\ 0 & 0 & I_4 & 0 & 0 & P0_2 & P1_2 \\ 0 & 0 & 0 & I_4 & 0 & P0_3 & P1_3 \\ 0 & 0 & 0 & 0 & I_4 & P0_4 & P1_4 \end{bmatrix}.$$

Here, I_4 is the identity matrix of size 4×4:

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The matrix P contains the parity symbols. In particular, $P0$ represents the horizontal parity, and $P1$ represents the diagonal parity. Each element of $P0$ and $P1$, such as $P0_i$ and $P1_i$, corresponds to the submatrix of row i in the parity matrix [10].

4.2.1 Encoding

In general, encoding can be performed with quadratic complexity in the code length by multiplying the input data matrix by a generator matrix. However, to make encoding simpler and more efficient in practice, the EVENODD code uses a structured method based entirely on XOR operations.

According to Blaum M., Brady J., Bruck J. and Menon J. [11], the EVENODD code uses two forms of redundancy to tolerate up to two erasures: horizontal redundancy and diagonal redundancy.

- Columns with indices 0 to $k-1$ are data disks
- Column k stores horizontal parity
- Column $k+1$ stores diagonal parity

Let $a_{i,j}$ represent the symbol at row i and column j, where columns $0, \ldots, k-1$ hold the data symbols. During computations, any reference to row indexed $k-1$ (the imaginary row) is treated as zero. The example that follows will show why this row is necessary.

An auxiliary value, called the diagonal syndrome S, is defined as:

$$S = \bigoplus_{t=1}^{k-1} a_{k-1-t,t} \,. \tag{1}$$

This value is the XOR of the symbols along a specific diagonal in the data array, running from row $k-2$, column 1 to row 0, column $k-1$. The parity of this diagonal determines whether the code uses even or odd parity for all other diagonals. If $S = 1$, then odd parity is applied; otherwise, even parity is used.

Then, for each row index l, where $0 \leq l \leq k-2$, the horizontal parity is computed as the XOR of all data symbols in that row:

$$a_{l,k} = \bigoplus_{t=0}^{k-1} a_{l,t} \tag{2}$$

The diagonal parity symbols are computed as:

$$a_{l,k+1} = S \oplus \left(\bigoplus_{t=0}^{k-1} a_{(l-t) \bmod k,t} \right). \tag{3}$$

This equation calculates the XOR of a diagonal that wraps around the array using modulo k, ensuring the correct diagonal parity is applied based on the syndrome S.

Thus, column k contains the horizontal parity for each row, and column $k+1$ holds the diagonal parity, adjusted to follow either even or odd parity as determined by equation (1) [11].

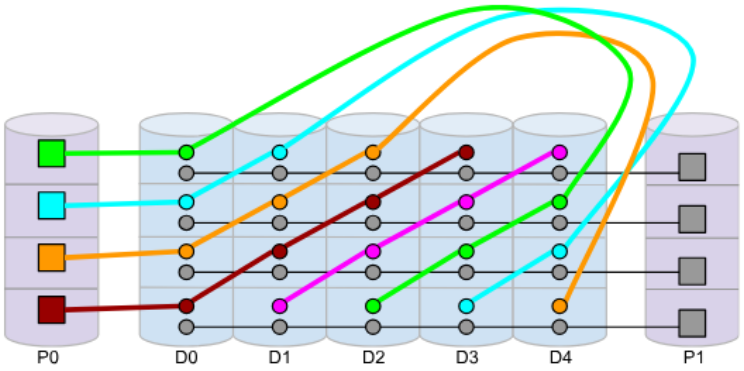

Figure 1. Parity relationships in a [7,5] EVENODD array.

Consider the case where $k = 5$ (as shown in Figure 1), and let the symbols be denoted by a_{ij} where $0 \leq i \leq 3, 0 \leq j \leq 6$. In this case, the redundant symbols are located in columns 5 and 6. A practical implementation can be modeled using 7 disks numbered from 0 to 6, where each disk contains 4 disk sectors. The data sectors are stored on disks numbered 0, 1, 2, 3, and 4, while the redundant sectors are stored on disks 5 and 6 (also labeled as $P1$ and $P0$ in Figure 1).

Equation (1) defines the encoding rule for the symbol S as:

$$S = a_{3,1} \oplus a_{2,2} \oplus a_{1,3} \oplus a_{0,4}$$

Following this, the horizontal redundant symbols for each row ℓ are calculated using:

$$a_{\ell,5} = a_{\ell,0} \oplus a_{\ell,1} \oplus a_{\ell,2} \oplus a_{\ell,3}, \oplus a_{\ell,4},$$

$$0 \leq \ell \leq 3.$$

The diagonal redundant symbols are computed as:

$$a_{0,6} = S \oplus a_{0,0} \oplus a_{4,1} \oplus a_{3,2} \oplus a_{2,3} \oplus a_{1,4}$$

$$a_{1,6} = S \oplus a_{1,0} \oplus a_{0,1} \oplus a_{4,2} \oplus a_{3,3} \oplus a_{2,4}$$

$$a_{2,6} = S \oplus a_{2,0} \oplus a_{1,1} \oplus a_{0,2} \oplus a_{4,3} \oplus a_{3,4}$$

$$a_{3,6} = S \oplus a_{3,0} \oplus a_{2,1} \oplus a_{1,2} \oplus a_{0,3} \oplus a_{4,4}.$$

Here, any term $a_{4,j}$ (from row 4) represents the imaginary row, which is treated as zero in calculations. These terms are often left out of the equations since XORing with zero doesn't change the result. However, they are included here for clarity to show the pattern.

This example follows the methods explained by Blaum et al. [11].

4.2.2 Decoding

The decoding process varies based on which disks (data or parity) fail. There are four cases according to [11]:

Case 1: $i = k, j = k + 1$:

Both redundant disks have failed. Recovery is equivalent to simply re-encoding:

- Use equation (2) to reconstruct the horizontal parity (disk k).
- Use equations (1) and (3) to reconstruct the diagonal parity (disk $k + 1$).

Case 2: $i < k$ and $j = k$

One data disk and the horizontal parity disk have failed.

First, calculate the intermediate diagonal parity value S from the surviving elements:

$$S = a_{(i-1) mod\ k, k+1} \oplus \left(\bigoplus_{l=0}^{k-1} a_{(i-l-1)\ mod\ k, l} \right).$$

For each row u, where $0 \leq u \leq k - 2$, use the value of S to recover the missing data symbol in column i (row u):

$$a_{u,i} = S \oplus a_{(i-1) mod\ k, k+1} \oplus \left(\bigoplus_{\substack{l=0 \\ l \neq i}}^{k-1} a_{(u+i-1)\ mod\ k, l} \right), 0 \leq u \leq k - 2$$

Once column i is reconstructed, the missing horizontal parity symbols $a_{u,k}$ are recovered using Equation (3).

Case 3: $i < k$ and $j = k + 1$

One data disk and the diagonal parity disk have failed.

- First, recover the data disk i using horizontal parity equation (2).
- Then, reconstruct the diagonal parity disk using equations (1) and (3).

Case 4 (Main Case): $i < k\ and\ j < k$

Both failed disks are data disks.

These cannot be recovered using only row or diagonal parity independently. Additionally, the special diagonal parity value S cannot be calculated using equation (2) since i and j are unavailable. To resolve this, Blaum et al. [11] introduced an alternative method for computing S using the XOR of all parity symbols from both of the parity disks:

$$S = \left(\bigoplus_{l=0}^{k-2} a_{l,k} \right) \oplus \left(\bigoplus_{l=0}^{k-2} a_{l,k+1} \right). \tag{4}$$

Once S is known, the horizontal and diagonal parities can be computed:

- Horizontal parity: XOR surviving symbols in each row using equation (2).
- Diagonal parity: Use diagonal parity S from equation (4) and surviving diagonals.

With these equations, a small linear system over GF(2) is formed. Solving this system allows recovery of all missing symbols in the two failed disks.

Example:

Consider the case where $k = 5$, and data disks $i = 0, j = 2$ have been erased:

$$\begin{bmatrix} ? & 0 & ? & 1 & 0 & 1 & 0 \\ ? & 1 & ? & 0 & 0 & 0 & 0 \\ ? & 1 & ? & 0 & 0 & 0 & 1 \\ ? & 1 & ? & 1 & 1 & 1 & 0 \end{bmatrix}.$$

First, the row parities from the surviving symbols can be calculated:

$$P_i = a_{i,0} \oplus a_{i,1} \oplus a_{i,2} \oplus a_{i,3} \oplus a_{i4}.$$

Compute each row:

- Row 0: $1 = a_{0,0} \oplus 0 \oplus a_{0,2} \oplus 1 \oplus 0 \;\Rightarrow\; a_{0,0} \oplus a_{0,2} = 0 \;\Rightarrow\; a_{0,0} = a_{0,2.}$
- Row 1: $0 = a_{1,0} \oplus 1 \oplus a_{1,2} \oplus 0 \oplus 0 \;\Rightarrow\; a_{1,0} \oplus a_{1,2} = 1 \;\Rightarrow\; a_{1,0} = a_{1,2} \oplus 1.$
- Row 2: $0 = a_{2,0} \oplus 1 \oplus a_{2,2} \oplus 0 \oplus 0 \;\Rightarrow\; a_{2,0} \oplus a_{2,2} = 1 \;\Rightarrow\; a_{2,0} = a_{2,2} \oplus 1.$
- Row 3: $1 = a_{3,0} \oplus 1 \oplus a_{3,2} \oplus 1 \oplus 1 \;\Rightarrow\; a_{3,0} \oplus a_{3,2} = 0 \;\Rightarrow\; a_{3,0} = a_{3,2}.$

Compute S from Equation (4) using columns 5 and 6:

$$S = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 1.$$

The diagonal parity Q can be calculated using equation (3):

$$Q_i = S \oplus a_{i,0} \oplus a_{i-1,1} \oplus a_{i-2,2} \oplus a_{i-3,3} \oplus a_{i-4,4}.$$

Using known values:

- $Q_0 = 0 = 1 \oplus a_{0,0} \oplus 0 \oplus a_{3,2} \oplus 0 \oplus 0 \Rightarrow a_{0,0} \oplus a_{3,2} = 1$
- $Q_1 = 0 = 1 \oplus a_{1,0} \oplus 0 \oplus 0 \oplus 1 \oplus 0 \Rightarrow a_{1,0} = 0$
- $Q_2 = 1 = 1 \oplus a_{2,0} \oplus 1 \oplus a_{1,1} \oplus 0 \oplus 1 \Rightarrow a_{2,0} \oplus a_{1,1} = 0 \Rightarrow a_{2,0} = a_{1,1}$
- $Q_3 = 0 = 1 \oplus a_{3,0} \oplus 1 \oplus a_{1,2} \oplus 1 \oplus 0 \Rightarrow a_{3,0} \oplus a_{1,2} = 1$

Final step is solving the system of equations.

a. $a_{0,0} = a_{0,2}$
b. $a_{1,0} = a_{1,2} \oplus 1$
c. $a_{2,0} = a_{2,2} \oplus 1$
d. $a_{3,0} = a_{3,2}$
e. $a_{0,0} \oplus a_{3,2} = 1$
f. $a_{1,0} = 0$
g. $a_{2,0} \oplus a_{0,2} = 0$
h. $a_{3,0} \oplus a_{1,2} = 1$

From (f), $a_{1,0} = 0$, so from (b), $a_{1,2} = 1$.

From (h), $a_{3,0} = 0$, and (d) gives $a_{3,2} = 0$.

From (e), $a_{0,0} = 1$, so from (a) $a_{0,2} = 1$.

From (g), $a_{2,0} = 1$ and from (c), $a_{2,2} = 0$.

The fully recovered matrix is:

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix},$$

which concludes the recovery of the missing data disks [11].

4.2.3 Performance

EVENODD codes are designed to tolerate two disk failures while relying only on XOR operations. They are computationally efficient, require no special hardware, and can be implemented using standard RAID-5 or RAID-6 controllers.

Encoding requires approximately $(2n-1)(n-1)$ XORs per block, which makes it an average complexity of $O(n^2)$.

Update performance depends on whether the modified symbol contributes to the diagonal parity (syndrome S). In the majority of updates, only 3 XORs are required, which is optimal. However, in about $\frac{1}{n-1}$ of the cases, when the symbol affects the diagonal parity, up to $n+1$ XORs are needed, making updates less efficient in those rare cases.

Decoding performance depends on the failure pattern. If only one data disk fails, recovery is fast and requires simple XORs across each row. However, if two data disks fail and syndrome S must be used, decoding becomes serialized: the missing symbols have to be recovered one by one in a fixed order, increasing latency and reducing efficiency. This scenario is less optimal than simpler failure cases.

Overall, EVENODD offers excellent encoding speed, especially in systems with large stripe sizes, and low computational complexity compared to Reed-Solomon codes, which rely on Galois field multiplications. Its main drawback is the occasional high update cost in specific configurations [10].

4.3 LDPC codes

Low-Density Parity-Check (LDPC) codes, introduced by R. Gallager in the 1960s and revived by D. MacKay in 1995, are known for their sparse parity-check matrices that enable efficient, iterative decoding. Their strong error-correcting performance and suitability for parallel processing have made them a key component in both modern communication standards and data storage systems [7].

4.3.1 Structure and Properties

In LDPC codes, the structure of the code is defined by a sparse parity-check matrix H, which is of size $r \times n$. The term sparse refers to the fact that the matrix contains significantly more zeros than ones.

A binary linear $[n, k]$ code is called a (J, K)-regular LDPC code if its parity-check matrix H has exactly J ones in each column and K ones in each row. If the number of ones is not the same in every row or column, the code is called an irregular. These values J and K are usually much smaller than the total number of columns and rows, which keeps the matrix sparse and helps make decoding faster and more efficient.

The parity-check matrix H of a regular LDPC code can be expressed as a vertical concatenation of submatrices:

$$H = \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_J \end{bmatrix}.$$

Each submatrix H_j, where $j = 1, 2, \ldots, J$ is a column permutation of H_1, consisting of columns with weight 1 and rows with weight K. This structure results in a (J, K)-regular code, where each codeword symbol participates in exactly J parity checks and each check equation involves K symbols [7].

4.3.2 Matrix Representation and Tanner Graph

The following shows the parity-check matrix H of a $(2,4)$-regular $[8,4]$ LDPC code, along with its corresponding Tanner graph (Figure 2).

$$H = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}.$$

Each row of H defines a parity-check equation (check node c_i) and each column corresponds to a code symbol (variable node v_j). This structure can be shown as a Tanner graph, a bipartite graph introduced by Tanner R. M. in 1981 [12]. In this graph, symbol nodes and check nodes

are connected by edges whenever $H[i][j] = 1$, indicating that the j-th symbol participates in the i-th parity-check equation.

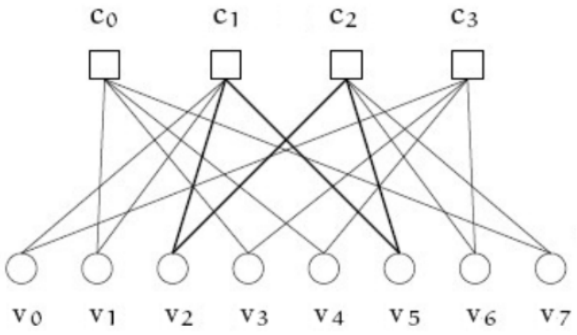

Figure 2. Tanner graph of (2, 4) regular [8, 4] LDPC code

The Tanner graph shows how variable nodes (code symbols) connect to check nodes (parity constraints). The connections between them correspond to the 1s in the parity-check matrix H. To satisfy all constraints, we have to solve:

$$\begin{cases} v_1 + v_3 + v_4 + v_7 = 0 \quad (c_0) \\ v_0 + v_1 + v_2 + v_5 = 0 \quad (c_1) \\ v_2 + v_5 + v_6 + v_7 = 0 \quad (c_2) \\ v_0 + v_3 + v_4 + v_6 = 0 \quad (c_3) \end{cases}.$$

This structure also forms the basis for the peeling decoder, which exploits the sparsity and graph structure to iteratively recover erased bits, as explained more in the next chapter [7].

4.3.3 Encoding and decoding

Like any linear code, LDPC codes can be encoded using generator matrix multiplication. However, since the generator matrix G is typically dense, this would require $O(n^2)$ operations. Instead, practical implementations exploit a structured parity-check matrix H, such as a lower or upper bidiagonal form, enabling iterative encoding through simple recursive operations on parity bits, reducing complexity to $O(n)$ [7].

LDPC codes can always be decoded using Gaussian elimination, but this has worst-case complexity of $O(n^3)$. The decoding can be more efficiently performed using the so-called peeling decoder described by Luby M. G. et al. [13], which achieves linear complexity $O(n)$ for erasure channels. The peeling decoder operates directly on the parity-check matrix. The decoder iteratively searches for parity-check equations (rows of H) that contain exactly one

unknown symbol. When such equation is found, the unknown can be uniquely determined from the known symbols. Once recovered, the value is substituted into other equations, potentially revealing new equations with only one unknown. This process continues until all erased symbols are recovered or no solvable equations remain. The sparsity of H ensures that each step involves only a small number of operations, and the total decoding complexity grows linearly with the code length [13].

Example: Consider the (2,4)-regular [8,4] LDPC code discussed earlier. Suppose the symbols x_0, x_1, x_2 are erased. At first, none of the equations has exactly one unknown symbol. However, if equation 3: $x_2 + x_5 + x_6 + x_7 = 0$ becomes solvable because x_5, x_6, x_7 are known, then x_2 can be recovered. Next, we substitute x_2 into equation 2, which now has two unknowns left. However, equation 1 has only x_1 unknown, so we can solve for x_1. Finally, knowing x_1, we return to equation 2 and solve for x_0. This illustrates the iterative nature of decoding.

4.3.4 Performance

LDPC codes are not MDS, but they get very close to MDS-level performance as the number of devices increases. They allow much faster encoding and decoding compared to traditional MDS codes like Reed-Solomon. Unlike MDS codes, the exact minimum distance d of LDPC codes is generally unknown and difficult to compute, but it does not play a significant role in their iterative decoding. Instead, performance depends more on the code rate and the sparsity of the parity-check matrix. Both encoding and decoding have a time complexity of $O(n)$, where n is the total number of disks.

Updates are also efficient, requiring only $O(1)$ time per updated symbol. This is possible because each symbol is connected to only a small number of parity checks, thanks to the sparse design of the parity-check matrix.

LDPC codes need slightly more redundancy than MDS codes to achieve similar fault tolerance, but the extra overhead is small. It is usually less than 10% when there are more than 10 disks. LDPC codes offer encoding speeds that are 40% to 60% faster than optimal MDS codes. This makes them a practical choice for large storage systems or systems where speed and low delay are important [10].

4.4 Convolutional codes

Convolutional codes are error-correcting codes designed for data transmitted or stored as a continuous stream. Unlike block codes, which encode independent, fixed-length blocks, convolutional codes generate output bits that depend not only on the current input but also on previous inputs. This "memory" adds redundancy over time.

A typical convolutional encoder uses shift registers to store past input bits. At each step, it combines the current input bit with selected previous bits to produce the output. For example, in a rate $R = \frac{1}{2}$ encoder, the input is a stream of bits such as $u_0, u_1, u_{2,\dots}$, and so on. For every input bit, the encoder produces two output bits based on the current bit and several prior ones stored in memory [14].

4.4.1 Encoding

The encoder of a convolutional code is defined by its generator polynomials, which describe how the input bits are combined to produce the output bits over time. These polynomials are often used to represent the encoding process, but in practice, this is typically expressed in terms of a generator matrix.

The generator matrix is commonly expressed using the delay operator D, where D represents a one-time-unit delay. For example, a systematic encoder with rate $R = \frac{3}{4}$, memory $M = 2$, and three input bits per block can be described by the generator matrix:

$$G(D) = \begin{bmatrix} 1 & 0 & 0 & 1 + D + D^2 \\ 0 & 1 & 0 & 1 + D^2 \\ 0 & 0 & 1 & 1 + D \end{bmatrix},$$

where each row defines the logic for computing the parity-bits.

To encode a message, it is multiplied by the generator matrix.

Convolutional codes can also be described using a sliding window parity-check matrix H. This matrix is composed of smaller submatrices $H_0, H_1, \dots, H_M$, where M is the memory of the code and each H_i has dimensions $(n - k) \times n$, with n being the number of bits per encoded block and k the number of input bits per block. These submatrices are arranged in a staircase-like structure:

$$\begin{bmatrix} H_2 & H_1 & H_0 & 0 & \cdots & & & & \\ \ddots & \ddots & \ddots & \ddots & \ddots & & & & \\ \cdots & 0 & H_2 & H_1 & H_0 & 0 & \cdots & & \\ & \cdots & 0 & H_2 & H_1 & H_0 & 0 & \cdots & \\ & & \cdots & 0 & H_2 & H_1 & H_0 & 0 & \cdots \\ & & & & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & & \cdots & 0 & H_2 & H_1 & H_0 \end{bmatrix}.$$

This structure represents how parity bits depend on both current and past input blocks, illustrating the code's memory and sequential operation [14].

4.4.2 Decoding

The peeling decoder can be applied to convolutional codes by solving the constraints defined by the parity-check matrix $H(D)$. For example, a single-row convolutional parity-check matrix can be defined as:

$$H(D) = [1 + D + D^2 \;\; 1 + D^2 \;\; 1 + D \;\; 1].$$

Each entry in $H(D)$ represents how a particular symbol at a given time is involved in parity checks with past symbols.

In storage systems, erasures correspond to lost symbols or blocks, and the decoder iterates over sparse parity-check equations to recover these lost symbols. Each step solves for one missing symbol, and the process continues until all missing symbols are found or no more solutions are possible. The sparsity of $H(D)$ makes this process efficient [14].

4.4.3 Performance

Encoding is typically performed iteratively with shift registers, making it linear in complexity relative to block length. Generally, for decoding, complexity depends on the number of erasures corrected (ν), with worst-case complexity $O(\nu^3)$. However, the peeling decoder can also be applied here allowing for a linear-time complexity in many practical scenarios [7].

5. Storage Systems

Modern storage systems balance performance, cost, and fault tolerance. Traditional systems (DAS/NAS) rely on replication or RAID, while distributed systems adopt erasure coding for high durability with lower storage overhead. This section categorizes storage architectures, analyzes redundancy mechanisms, and highlights where EC is most impactful.

5.1 Storage System Models

A storage system consists of hardware and software components that store, manage, and protect data. It includes storage devices (e.g., HDDs, SSDs), controllers for data access, and software for organizing the information. Storage systems range from simple direct-attached solutions to complex distributed and cloud-based architectures [15].

5.1.1 Direct-Attached Storage (DAS)

Direct-Attached Storage (DAS) connects storage devices directly to a single host, either internally or externally, with no network involved. Only the host can access the data, and other devices must go through it. Examples include internal HDDs and SSDs and external drives via USB or SATA. In enterprise setups, servers may connect directly to storage arrays, but access goes still through host. RAID can be implemented via software on the host to combine drives for better reliability or performance, as discussed later in Chapter 5.2 [16].

DAS is simple to deploy and manage, making it popular for small to medium enterprises. However, it has scalability limits due to port and bandwidth restrictions. Storage capacity cannot be easily shared across multiple hosts, which may lead to uneven resource utilization and decreased service availability as capacity limits are approached. Despite these challenges, DAS often delivers better performance compared to networked storage by eliminating network latency [16].

5.1.2 Network-Attached Storage (NAS)

Network-Attached Storage (NAS) is a file-level storage system connected to a local area network (LAN), providing centralized storage accessible by multiple clients. Unlike DAS, NAS operates independently of any single host, allowing simultaneous data access from various devices and users.

This makes NAS ideal for environments requiring shared data access, such as collaborative offices, media storage, and backup solutions. NAS devices run dedicated operating systems

with specialized storage controllers, often including hardware RAID controllers. RAID configurations are common in NAS, but unlike DAS, the RAID is managed internally by the NAS controller rather than the host operating system [16].

5.1.3 Distributed Storage Systems (DSS)

Distributed Storage Systems (DSS) are architectures that store data across multiple machines. Widely used by cloud providers and big data platforms, DSS employ erasure coding instead of simple replication to protect data and optimize storage.

Table 2 summarizes popular distributed storage systems, their common erasure codes, typical data block sizes, and relevant notes.

Table 2. Summary of distributed storage systems, erasure codes, and typical data sizes.

Storage System	Erasure Code(s)	Data Block Size	Notes
HDFS	RS (6,3), RS (3,2), RS (10,4), EVENODD (2,1)	128 MB (per data block)	Originally 3x replication, now mostly erasure coding
Ceph	RS (16,10), RS (6,4)	Varies, often 4-128 MB	Scales to exabyte-level (EB) storage
Microsoft Azure Blob Storage	LRC (6, 2, 2)	Varies, often 128 MB	Non-MDS, optimized for repair efficiency
Amazon S3	RS codes	Varies	Exact code parameters unknown

The information in this chapter and (Table 2) is compiled from a range of sources: [17], [18], [19], [20].

5.2 RAID configuration

RAID[1] (Redundant Array of Independent Disks) combines multiple physical drives into a single logical unit to improve performance, reliability, or both. While RAID configurations are primarily designed to offer redundancy, the degree of data protection varies across different RAID levels. For example, RAID 0 uses striping to boost performance but provides no redundancy, meaning a single disk failure results in total data loss. In contrast, RAID 1 uses mirroring to duplicate data across drives, offering fault tolerance, but does not use parity or improve storage efficiency. Other RAID levels include parity to balance redundancy and storage overhead, allowing recovery from one or more disk failures [21].

In the following section, based on Plank J. S. [8] explanation, the focus will be on RAID-6, which is widely used in smaller-scale systems due to its ability to tolerate the failure of any two disks while maintaining a good balance between performance and storage efficiency. RAID-6 uses erasure codes, specifically simple XOR and Reed-Solomon codes. This makes it well-suited for individual servers or smaller storage arrays, where the probability of more than two disk failures occurring simultaneously is relatively low.

RAID-6 systems use two parity disks, P and Q, to achieve fault tolerance. Each disk block is treated as a vector of bytes, and parity is calculated independently for each byte position.

The P parity disk stores sector-wise XOR values across all data disks. That is, each sector P_i contains:

$$P_i = D_{0,i} \oplus D_{1,i} \oplus \ldots \oplus D_{n-1,i}$$

where $D_{j,i}$ is the i-th sector of the j-th data disk, and addition is performed over the field GF(2^8). The Q disk provides a second, independent parity by using Galois field multiplication. It follows a Reed-Solomon code construction, where the i-th sector of Q is:

$$Q_i = \alpha^0 \cdot D_{0,i} \oplus \alpha^1 \cdot D_{1,i} \oplus \ldots \oplus \alpha^{n-1} \cdot D_{n-1}, i.$$

Here, α is a primitive element of the field GF(2^8).

While computing the P disk requires only XOR operations and is computationally lightweight, the Q disk involves repeated multiplications in the field GF(2^8), which are more costly.

[1] The term was introduced in the original RAID taxonomy paper: D. A. Patterson, G. A. Gibson, and R. H. Katz, A Case for Redundant Arrays of Inexpensive Disks (RAID), 1988 [22].

Standard CPUs do not natively support GF multiplication, making direct computation inefficient. To speed up these operations, precomputed lookup tables are often used to replace runtime calculations with simple table accesses [23].

In modern RAID-6 implementations, the process of calculating the Q redundancy values is optimized by simplifying the computation. Instead of performing complex calculations, the system reorganizes the process to only use repeated multiplications by a fixed constant, typically the number 2. This makes the calculation easier and faster. Additionally, the system can take advantage of CPU vector instructions, such as SSE[2] and AVX[3], which allow it to process multiple bytes in parallel. This parallel processing significantly boosts the efficiency of the Q parity computation [23].

[2] SSE (Streaming SIMD Extensions): SIMD instruction set for parallel data processing, supported by Intel and AMD CPUs [24].

[3] AVX (Advanced Vector Extensions): Extends SIMD capabilities, enabling wider vector operations for improved performance [24].

6. Comparing Erasure Codes in Practice

This section compares the four erasure codes described earlier, focusing on how they perform and where they are best used in storage systems.

In large-scale storage systems, even small disk failure probabilities become significant. For instance, in a system with 10,000 disks, each with a mean time to failure (MTTF) of one million hours, the chance of at least one failure at any given time is nearly 1%. To protect data in such environments, erasure coding offers more storage-efficient fault tolerance compared to simple replication [25].

RS codes are space-optimal and highly fault-tolerant, widely used in distributed storage due to their reliability. However, in practice, they rely on small finite fields (e.g., GF(2^4), GF(2^8)), which limits block length and forces data to be split into many short codewords. This increases storage overhead to maintain recovering reliability. Longer blocks reduce overhead but significantly raise encoding and decoding complexity and latency. RS codes balance storage efficiency and reliability well but can suffer throughput issues in large distributed systems with short blocks [26].

EVENODD codes are designed for disk array systems like RAID, where fast encoding/decoding with low computational cost is essential. They only recover from up to two erasures, which limits their fault tolerance compared to RS and LDPC codes. Their key advantage is the exclusive use of XOR operations, which are trivial to implement in hardware controllers, making EVENODD extremely efficient for hardware-accelerated environments. However, this limited fault tolerance makes EVENODD less suitable for large-scale distributed storage systems where multiple simultaneous failures are common [27].

LDPC codes provide fast encoding and decoding with low storage overhead by using sparse parity-check matrices. They can handle multiple failures and scale well in distributed and cloud storage systems. Although LDPC codes are not strictly MDS, they approach the MDS property as the code length n grows large. This makes them an attractive choice for large-scale systems where fault tolerance and computational efficiency are both important [28].

Convolutional codes are typically designed for streaming data, where input is treated as an infinite sequence rather than a fixed-size block. As such, they are not commonly used in storage systems, which generally operate on finite data blocks [7]. However, a block-based version of convolutional codes, suitable for storage-like applications, is introduced in chapter 7. This

allows convolutional codes to be adapted for use in distributed storage, offering near-optimal fault tolerance with linear-time encoding and decoding when using iterative encoding and peeling decoder.

Table 3. Overview of erasure code characteristics and use cases

Code Type	Fault tolerance	Encoding complexity	Decoding complexity	Storage overhead	Typical use case
Reed-Solomon	Maximum (MDS)	$O(n^2)$	$O(n^3)$	Low (optimal)	Large-scale cloud, archival storage
EVEODD	Two disk failures	$O(n^2)$	Depends, (worst $O(n^2)$)	Moderate	Small systems, RAID
LDPC	Near-MDS, scalable	$O(n)$	$O(n)$	Slightly higher than MDS	Large distributed systems
Convolutional	Near optimal	$O(n)$	$O(n)$	Moderate	Distributed storage systems

Choosing the best erasure code depends on practical needs. In real storage systems, disk access times are relatively slow, so extra encoding and decoding time is often negligible. However, storage overhead is costly at scale. Reed-Solomon codes, with their strong fault tolerance, storage efficiency, and well-established implementations, remain the most widely used in distributed storage today [25].

7. Adapting Convolutional Codes for Low-Complexity Storage Systems

This chapter describes the design and implementation of a convolutional encoder and two decoding algorithms for storage systems. The decoders are evaluated and compared based on their ability to recover from erasures under varying erasure probabilities.

7.1 Sliding-Window Encoder

The encoder[4] implemented in this work is a rate $R = 3/4$ sliding-window convolutional encoder designed for erasure-resilient data storage. It processes the input message in fixed-size blocks of three bits and generates a four-bit codeword by appending one parity bit to each message. The encoder has memory $M = 2$, meaning that each output depends on the current input message as well as the two previous input messages.

To compute the parity bit, three binary row vectors H_0, H_1, and H_2 are used. These represent the contribution of the current input, the previous input, and the input from two time steps ago, respectively. For each time step i, the parity bit p_i is calculated as:

$$p_i = H_0 \cdot m_i^T + H_1 \cdot m_{i-1}^T + H_2 \cdot m_{i-2}^T \mod 2$$

where m_i is the current 3-bit input message, and m_{i-1}, m_{i-2} are the preceding input messages.

In this implementation:

$$H_0 = [1\ 1\ 0],\ H_1 = [1\ 0\ 1],\ H_2 = [1\ 1\ 1].$$

The encoded codeword et each time step is:

$$c_i = [m_i, p_i],$$

which includes the original 3-bit message followed by one computed parity bit [14].

[4] The source code for the encoder and decoder implementations is available at [29].

7.2 Sliding-Window Decoder via Linear System Solving

This decoder operates on a sliding window over the received codeword, correcting erasures by solving a system of linear equations derived from the parity-check matrix H. The decoding is based on the condition that any valid codeword must satisfy:

$$\boldsymbol{y}H^T = 0.$$

When some symbols are erased, the equation is solvable from:

$$\boldsymbol{z}H^T_{I(e)} = \boldsymbol{s}(e).$$

Here, $I(e)$ denotes the indices of the erased symbols, $\boldsymbol{z}$ represents the unknown erased bits, and $\boldsymbol{s}(e)$ is the syndrome computed from the known symbols.

At each window position, the decoder:

1. Computes the syndrome $\boldsymbol{s}(e)$ using the known values in the window.
2. Extracts the submatrix H_{sub} corresponding to the erased positions.
3. Solves the system $H_{sub} \cdot \boldsymbol{z} = \boldsymbol{s}(e)\ mod\ 2$, assuming the submatrix has full rank.

Recovered values are written back into the window, and the process continues for the entire input. After decoding, the original message is reconstructed by taking only the first message bits from each codeword [14].

7.3 Peeling Decoder

The peeling decoder is an iterative algorithm designed for efficient erasure recovery when the parity-check matrix H is sparse. It repeatedly scans each row of H (representing parity-check equations) and identifies cases where only a single unknown bit remains in the equation. In such cases, the missing bit can be directly recovered by enforcing the parity constraint. This process continues until no more erasures can be resolved. The method is computationally lighter, than solving the system of linear equations. Once all recoverable bits are decoded, the original 3-bit message portions are extracted to reconstruct the message [13].

7.4 Comparing Decoders

In this section, the performance of the two implemented decoders, sliding-window decoder based on solving linear systems, and the peeling decoder is compared. The metric used here is the Frame Erasure Rate (FER) as a function of the erasure probability. Both decoders were tested under identical conditions using the same convolutional code parameters.

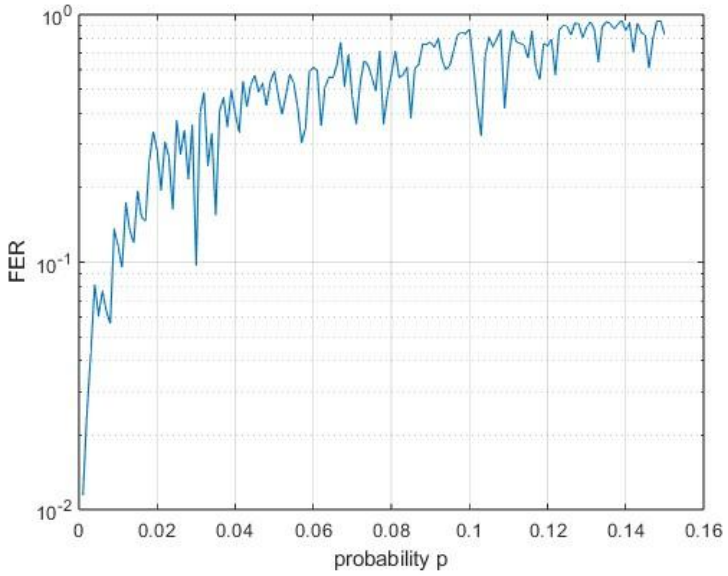

Figure 3. FER vs. Erasure Probability (Sliding-Window Decoder)

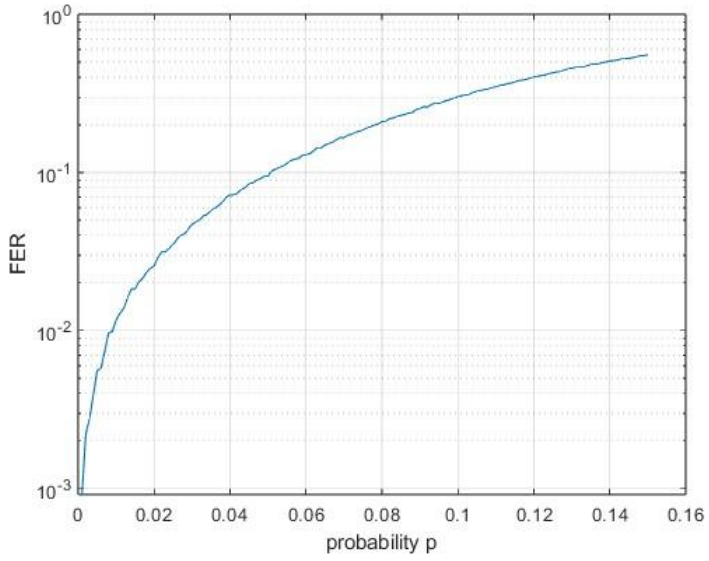

Figure 4. FER vs. Erasure Probability (Peeling Decoder)

The results show that the peeling decoder performs better than the sliding-window decoder at all erasure probabilities. The sliding-window decoder gives less stable results and shows more variation in performance. In contrast, the peeling decoder has a lower and more consistent Frame Erasure Rate (FER). This is because it makes better use of the sparse structure of the parity-check matrix and does not need to solve full systems of equations. Overall, the peeling decoder is a better choice for simple and reliable data recovery in low-complexity storage systems.

8. Conclusion

This thesis explored the design, implementation, and practical considerations of various erasure codes used in storage systems, with a particular focus on Reed-Solomon, EVENODD, LDPC, and convolutional codes. Each of these codes offers different trade-offs between fault tolerance, computational complexity, and storage overhead, making them suitable for different use cases.

Reed-Solomon codes, being MDS, provide optimal fault tolerance with minimal storage overhead and remain the industry standard for distributed and archival storage. However, their computational complexity makes them less ideal for latency-sensitive or resource-constrained environments. XOR-based EVENODD codes are efficient for small-scale RAID systems, but their limited erasure correction capability restricts their applicability in large-scale distributed environments.

LDPC codes are a scalable alternative, offering low encoding/decoding complexity and near-MDS performance, especially effective in large cloud systems where computational efficiency and fault tolerance are equally critical. Convolutional codes, although traditionally used in streaming applications, were adapted in this work for block-based storage contexts. Their iterative encoding and sparse parity structure enable low-complexity decoding via the peeling algorithm, making them promising candidates for lightweight storage systems.

The best choice depends on system needs. While RS remains common, LDPC and convolutional codes offer good alternatives for scalable and lightweight storage.

References

[1] Plank J. S. Erasure Codes for Storage Systems. 2013. https://www.usenix.org/system/files/login/articles/10_plank-online.pdf (accessed 25.12.2024).

[2] Plank J. S., Luo J., Xu L. An Efficient XOR-Scheduling Algorithm for Erasure Codes Encoding. 2009. https://web.eecs.utk.edu/~jplank/plank/papers/DSN-2009.pdf (accessed 25.12.2024).

[3] Vinayak R. K. Erasure Coding for Big-data Systems: Theory and Practice. 2016.

[4] Roth R. M. Introduction to Coding Theory. 2007.

[5] Khan Academy. XOR Bitwise Operation. https://www.khanacademy.org/computing/computer-science/cryptography/ciphers/a/xor-bitwise-operation (accessed 25.12.2024).

[6] Cook J. D. MDS codes. 2020. https://www.johndcook.com/blog/2020/03/07/mds-codes/ (accessed 27.12.2024).

[7] Bocharova I., Kudryashov B. Practical Error Correcting Coding. 2025.

[8] Plank J. S., A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. 1999.

[9] Riley M., Richardson I. Reed-Solomon Codes. 1998. https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html (accessed 20.12.2024).

[10] Plank J. S. T1: Erasure Codes for Storage Applications. 2005. https://web.eecs.utk.edu/~jplank/plank/papers/FAST-2005.pdf (accessed 25.12.2024).

[11] Blaum M., Brady J., Bruck J., Menon J. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. 1995.

[12] Tanner R. M. A Recursive Approach to Low Complexity Codes. 1981.

[13] Luby M. G., Mitzenmacher M., Shokrollahi M. A., Spielman D. A. Efficient Erasure Correcting Codes. IEEE Transactions on Information Theory. 2001.

[14] Bocharova I., Kudryashov B., Lyamin N., Frick E., Rabi M., Vinel A. Low Delay Inter-Packet Coding in Vehicular Networks. Future Internet. 2019. https://www.mdpi.com/1999-5903/11/10/212?type=check_update&version=2 (accessed 5.05.2025).

[15] Thomasian A. Storage Systems: Organization, Performance, Coding, Reliability, and Their Data Processing. 2022.

[16] EMC Corporation. Information Storage and Management. 2009. https://referenceglobe.com/CollegeLibrary/library_books/20180227080126ISM%20(1).pdf (accessed 27.04.2025).

[17] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html (accessed 11.05.2025).

[18] Red Hat. Red Hat Ceph Storage Documentation: Chapter 4. Ceph erasure coding. https://docs.redhat.com/en/documentation/red_hat_ceph_storage/8/html/edge_guide/ceph-erasure-coding_edge#ceph-erasure-coding_edge (accessed 11.05.2025).

[19] Huang C., Simitci H., Xu Y., Ogus A., Calder B., Gopalan P., Li J., Yekhanin S. Erasure Coding in Windows Azure Storage. Microsoft Corporation. https://www.usenix.org/system/files/conference/atc12/atc12-final181_0.pdf (accessed 10.05.2025).

[20] Warfield A. Building and Operating a Pretty Big Storage System Called S3. All Things Distributed. 2023. https://www.allthingsdistributed.com/2023/07/building-and-operating-a-pretty-big-storage-system.html (accessed 10.05.2025).

[21] Khawatreh S., El-Omari N. K. T. RAID-based Storage Systems. 2018.

[22] Patterson D. A., Gibson G. A., and Katz R. H., A Case for Redundant Arrays of Inexpensive Disks (RAID), 1988. https://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-391.pdf (accessed 25.04.2025).

[23] Anvin H. P The mathematics of RAID-6. 2011.

[24] Jeong H., Kim S., Lee W., Myung S.-H. Performance of SSE and AVX Instruction Sets. 2012. https://arxiv.org/abs/1211.0820 (accessed 28.04.2025).

[25] Xin Q. Understanding and Coping with Failures in Large-Scale Storage Systems. Storage Systems Research Center. 2005. https://ssrc.us/media/pubs/0061f15bed89da16e199ad92c4036aef8c7bd1b0.pdf (accessed 10.05.2025).

[26] Arslan S. S. Reed-Solomon (RS) vs Fountain Codes in Distributed Storage: A High-level Overview. 2018. https://www.linkedin.com/pulse/reed-solomon-rs-vs-fountain-codes-distributed-storage-suayb-s-arslan (accessed 15.05.2025).

[27] Tao W. Application and Performance Analysis of EVENODD in RAID Architecture. 2024. https://drpress.org/ojs/index.php/HSET/article/view/19050/18614 (accessed 15.05.2025).

[28] Vairaperumal B. Tharini C. Review on LDPC Codes for Big Data Storage. 2021.

[29] Pikani O. Convolutional_34. GitHub. 2025. https://github.com/OIivr/Convolutional_34

License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Oliver Pikani

1. grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis

   Comparative Analysis of Erasure Correcting Codes in Data Storage,

   supervised by Irina Bocharova;

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Oliver Pikani

15/05/2025