UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Proiect — Procesarea Semnalelor

COMPRESIA SERIILOR DE TIMP FOLOSIND ALGORITMUL GORILLA

Dima Cristian-Mihai

Muscalu David-Cristian

Coordonator științific
Prof. Dr. Cristian Rusu

București, 2026

Rezumat

Sistemele moderne de monitorizare generează volume imense de date sub formă de serii de timp — secvențe ordonate de perechi *(timestamp, value)* care înregistrează evoluția în timp a diferitelor metrici. Stocarea eficientă a acestor date reprezintă o provocare semnificativă, deoarece abordarea naivă necesită 16 bytes pentru fiecare datapoint (8 bytes pentru timestamp și 8 pentru valoarea în format IEEE 754 double).

În acest proiect analizăm și implementăm algoritmul de compresie **Gorilla**, dezvoltat de Facebook pentru baza lor de date de serii de timp in-memory. Soluția propusă combină două tehnici complementare: *delta-of-delta encoding* pentru comprimarea timestamp-urilor și *XOR encoding* pentru comprimarea valorilor în virgulă mobilă. Ambele tehnici exploatează proprietățile specifice ale seriilor temporale — periodicitatea timestamp-urilor și corelația temporală a valorilor — pentru a obține rate de compresie semnificative.

Implementarea realizată de noi în Python demonstrează eficiența algoritmului pe două seturi de date: metrici de utilizare CPU (serie univariată) și date climatice dintr-un mediu interior (serie multivariată cu 8 variabile). Rezultatele experimentale arată rate de compresie de aproximativ 2–4x, reducând consumul de memorie de la 16 bytes la aproximativ 4–8 bytes per datapoint, în funcție de caracteristicile datelor.

Cuprins

Capitolul 1

Introducere

1.1 Contextul problemei

O **serie de timp** este o secvență ordonată de observații înregistrate la momente succesive de timp, de obicei la intervale regulate. Formal, o serie temporală poate fi definită ca o funcție:

$$X : T \to \mathbb{R}, \quad X = \{(t_1, x_1), (t_2, x_2), \ldots, (t_n, x_n)\} \tag{1.1}$$

unde $T \subseteq \mathbb{Z}$ reprezintă mulțimea timestamp-urilor (de regulă, în milisecunde sau secunde de la epoca Unix), iar $x_i \in \mathbb{R}$ reprezintă valoarea observată la momentul t_i.

Aplicațiile care generează serii de timp sunt omniprezente în infrastructura digitală contemporană, iar volumul de date generate de aceste sisteme este enorm. Pentru a ilustra scala problemei, să considerăm sistemul de monitorizare al companiei Facebook (Meta), descris în articolul original Gorilla [**gorilla2015**]:

- Peste **2 miliarde** de serii temporale unice;
- Aproximativ **12 milioane** de datapoints adăugate *în fiecare secundă*;
- Aceasta echivalează cu peste **1 trilion** de puncte pe zi;
- La 16 bytes per datapoint (8 pentru timestamp + 8 pentru valoare), rezultă un necesar de aproximativ **16 TB de RAM** zilnic.

Această rată masivă de ingestie a datelor face imposibilă stocarea necomprimată și impune dezvoltarea unor tehnici eficiente de compresie.

1.2 Motivația proiectului

Stocarea seriilor temporale prezintă provocări specifice care diferă de cele întâlnite în bazele de date tradiționale:

1.2.1 Cerințe de performanță

Sistemele de monitorizare necesită:

1. **Disponibilitate ridicată pentru scrieri**: Datele trebuie să poată fi scrise în orice moment, chiar și în prezența unor defecțiuni parțiale ale sistemului. O rată ridicată de succes pentru operațiile de scriere este esențială, deoarece pierderea datelor de monitorizare poate masca probleme critice.
2. **Latență redusă pentru citiri**: Interogările trebuie să returneze rezultate în milisecunde, nu secunde. Aceasta permite inginerilor să reacționeze rapid la anomalii și să diagnosticheze probleme în timp real.
3. **Prioritizarea datelor recente**: În contextul monitorizării, datele recente sunt mult mai valoroase decât cele vechi. Faptul că un serviciu a fost indisponibil acum 5 minute este mai relevant decât același eveniment petrecut acum o săptămână.

1.2.2 Particularități ale datelor

Seriile temporale de monitorizare au proprietăți distincte care pot fi exploatate pentru compresie:

1. **Timestamp-uri cvasi-periodice**: Majoritatea sistemelor de monitorizare colectează date la intervale regulate (de exemplu, la fiecare 60 de secunde). Deși pot exista mici variații (jitter) datorită latențelor de rețea sau încărcării sistemului, intervalul dintre timestamp-uri consecutive rămâne relativ constant.
2. **Corelație temporală a valorilor**: Valorile consecutive dintr-o serie temporală tind să fie similare. Temperatura într-o cameră nu sare brusc de la 22°C la 50°C; utilizarea CPU a unui server nu variază dramatic de la o secundă la alta în condiții normale.
3. **Toleranță la pierderi minore**: Spre deosebire de sistemele financiare sau medicale, sistemele de monitorizare pot tolera pierderea ocazională a câtorva puncte de date, atâta timp cât tendințele generale rămân vizibile.

1.3 Obiectivele proiectului

Prezentul proiect își propune:

1. **Înțelegerea și implementarea algoritmului Gorilla**: Studiul detaliat al tehnicilor de compresie delta-of-delta și XOR, precum și implementarea lor completă în Python.
2. **Suport pentru serii multivariate**: Extinderea algoritmului pentru a gestiona eficient serii cu mai multe variabile per timestamp.
3. **Validarea experimentală**: Testarea implementării pe seturi de date reale și măsurarea ratelor de compresie obținute, comparativ cu stocarea naivă.
4. **Documentarea completă**: Prezentarea detaliată a fundamentelor teoretice, a algoritmilor și a deciziilor de implementare.

Capitolul 2

State-of-the-Art

Înainte de a prezenta algoritmul Gorilla, vom trece în revistă provocările cărora bazele de date cu serii de timp încearcă să le facă față și compromisurile pe care le fac.

2.1 Baze de date pentru serii de timp (TSDB)

O **bază de date pentru serii temporale** (Time Series Database — TSDB) este un sistem software optimizat pentru stocarea și interogarea datelor indexate după timp. Spre deosebire de bazele de date relaționale tradiționale, TSDB-urile sunt proiectate pentru:

- Rate foarte mari de scriere (milioane de puncte pe secundă);
- Interogări pe intervale de timp („dă-mi toate valorile între ora 10:00 și 11:00”);
- Agregări temporale (medii, sume, percentile pe ferestre de timp);
- Retenție automată a datelor (ștergerea datelor mai vechi de X zile).

2.2 OpenTSDB

OpenTSDB (Open Time Series Database) este o bază de date distribuită pentru serii temporale, construită peste Apache HBase [**opentsdb**]. HBase, la rândul său, este un sistem de stocare distribuit bazat pe modelul BigTable al Google.

2.2.1 Arhitectură

OpenTSDB utilizează un model de date în care fiecare serie temporală este identificată printr-un set de perechi cheie-valoare numite *tag-uri*:

```
metric: sys.cpu.user
tags: {host=webserver01, cpu=0}
timestamp: 1458034800
value: 42.5
```

Datele sunt stocate în HBase într-o structură de tabel optimizată pentru scanări secvențiale pe intervale de timp.

2.2.2 Avantaje și limitări

Avantaje:

- Scalabilitate orizontală prin adăugarea de noduri HBase;
- Persistență durabilă pe disc cu replicare;
- Model de date flexibil cu tag-uri arbitrare.

Limitări:

- **Latență ridicată**: Fiind bazat pe disc, interogările pot dura secunde sau chiar minute pentru volume mari de date;
- **Complexitate operațională**: Necesită administrarea unui cluster HBase și Hadoop;
- **Nu este optimizat pentru date recente**: Toate datele sunt tratate egal, fără prioritizare a celor recente.

În contextul Facebook, OpenTSDB (prin HBase) nu putea scala pentru cerințele de latență necesare. Percentila 90 a timpului de interogare creștea de la câteva secunde la minute pe măsură ce volumul de date creștea.

2.3 Graphite și Whisper

Graphite este o suită de monitorizare compusă din trei componente: Carbon (serviciul de ingestie), Whisper (formatul de stocare) și Graphite Web (interfața de interogare) [**graphite**].

2.3.1 Formatul Whisper

Whisper utilizează un format de stocare *Round-Robin Database* (RRD). Caracteristicile principale:

- **Dimensiune fixă**: Fiecare serie temporală ocupă un spațiu fix pe disc, determinat la creare;
- **Intervale fixe**: Timestamp-urile sunt presupuse să vină la intervale exacte;
- **Suprascrierea datelor vechi**: Când spațiul se umple, datele noi le suprascriu pe cele mai vechi.

2.3.2 Limitări

- **Nu suportă jitter**: Dacă un punct de date vine la secunda 61 în loc de 60, trebuie fie ignorat, fie interpolat;
- **Stocare pe disc**: Similar cu OpenTSDB, latența este limitată de I/O-ul discului;

- **Fiecare serie într-un fișier separat**: La miliarde de serii, managementul fișierelor devine problematic.

2.4 Tehnici de compresie pentru serii de timp

Există multe abordări pentru compresia seriilor de timp, fiecare cu propriile dezavantaje:

2.4.1 Compresie cu pierderi (Lossy)

Tehnicile cu pierderi reduc volumul de date prin aproximare:

- **Downsampling**: Păstrarea doar a unui punct din N (de exemplu, un punct pe minut în loc de unul pe secundă);
- **Agregare**: Înlocuirea mai multor puncte cu media, suma sau alte statistici;
- **Piecewise Linear Approximation**: Aproximarea seriei cu segmente de dreaptă.

Aceste tehnici sunt utile pentru date istorice, dar inacceptabile pentru monitorizare în timp real, unde fiecare punct poate indica o problemă critică.

2.4.2 Compresie fără pierderi (Lossless)

Tehnicile fără pierderi păstrează exact datele originale:

- **Delta encoding**: Stocarea diferențelor între valori consecutive în loc de valorile absolute;
- **Run-length encoding (RLE)**: Compresia secvențelor de valori identice;

2.4.3 Compresie pentru numere în virgulă mobilă

Compresia valorilor *floating-point* prezintă următoarele provocări:

- Reprezentarea IEEE 754 distribuie informația pe 64 de biți (semn, exponent, mantisă);
- Diferențele mici între valori pot produce reprezentări binare foarte diferite;

Lindstrom și Isenburg [**lindstrom2006**] au propus o tehnică bazată pe predicție și codare XOR pentru compresia datelor științifice în virgulă mobilă. Gorilla adaptează această tehnică pentru streaming în timp real.

2.5 Poziționarea algoritmului Gorilla

Gorilla ocupă o nișă distinctă în peisajul TSDB-urilor:

Gorilla este gândit ca un **cache write-through** în fața unui TSDB persistent precum HBase.

Un write-through cache este un mecanism unde datele sunt scrise simultan în cache și în mediul de stocare permanent (back-end storage). Când o aplicație trimite un punct de date (un datapoint), acesta este stocat imediat în memoria RAM a lui Gorilla pentru interogări ultra-rapide și, în paralel, este trimis către un sistem persistent (precum HBase) pentru arhivare pe termen

Caracteristică	OpenTSDB	Graphite	Gorilla
Stocare	Disc (HBase)	Disc (Whisper)	Memorie
Latență citire	Secunde	Secunde	Milisecunde
Compresie	Minimă	Fără	Delta-of-delta + XOR
Jitter timestamp	Da	Nu	Da
Prioritizare date recente	Nu	Nu	Da
Persistență	Da	Da	Opțional (cache)

Tabela 2.1: Comparație între sisteme TSDB

lung. Cache-ul este mereu „la zi” cu baza de date. Dacă interogarea caută date recente, Gorilla le oferă instantaneu fără a mai accesa discul.

Aceste caracteristici au făcut din Gorilla sistemul preferat pentru monitorizare în timp real la Facebook, servind peste 85% din interogările către datele din ultimele 26 de ore.

Capitolul 3

Algoritmul Gorilla

Acest capitol prezintă în detaliu algoritmul de compresie Gorilla, fundamentele sale matematice și schemele de codare utilizate. Algoritmul este compus din două tehnici complementare: **delta-of-delta encoding** pentru timestamp-uri și **XOR encoding** pentru valori în virgulă mobilă.

3.1 Arhitectura generală

3.1.1 Structura datelor

În Gorilla, datele sunt organizate în **blocuri** de durată fixă (implicit 2 ore). Fiecare bloc conține:

1. Un **header** cu timestamp-ul de start al blocului (aliniat la granița de 2 ore);
2. O secvență de perechi *(timestamp, valoare)* comprimate;
3. Metadate despre numărul de puncte din bloc.

Alegerea blocurilor de 2 ore reprezintă un compromis între rata de compresie (blocuri mai mari = compresie mai bună) și granularitatea accesului (blocuri mai mici = citire mai rapidă pentru intervale scurte).

3.1.2 Fluxul de compresie

Schema generală de compresie este ilustrată în Figura 3.1.

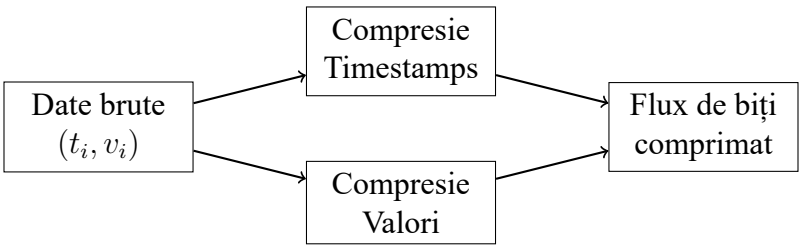

Figura 3.1: Fluxul general de compresie în Gorilla

3.2 Compresia Timestamp-urilor: Delta-of-Delta

3.2.1 Observația cheie

Timestamp-urile în sistemele de monitorizare sunt de obicei **cvasi-periodice**. Dacă un sistem colectează date la fiecare 60 de secunde, timestamp-urile consecutive vor fi:

$$t_0, t_0 + 60, t_0 + 120, t_0 + 180, \ldots \tag{3.1}$$

Chiar dacă există mici variații (de exemplu, $t_0 + 61$ în loc de $t_0 + 60$), diferența $\delta_i = t_i - t_{i-1}$ rămâne aproape constantă.

3.2.2 Delta encoding simplu

Prima idee ar fi să stocăm diferențele (delta-urile) dintre timestamp-ul curent și cel anterior în loc de valoarea timstamp-ului curent:

$$\delta_i = t_i - t_{i-1} \tag{3.2}$$

Pentru timestamp-urile de mai sus: $\delta_1 = 60, \delta_2 = 60, \delta_3 = 60, \ldots$

Astfel reducem valorile de stocat de la 64 de biți (timestamp absolut) la câțiva biți (delta mic). Dar putem obține o stocare a datelor și mai eficientă!

3.2.3 Delta-of-delta encoding

Gorilla utilizează **delta-of-delta** (diferența diferențelor):

$$D_i = \delta_i - \delta_{i-1} = (t_i - t_{i-1}) - (t_{i-1} - t_{i-2}) \tag{3.3}$$

Pentru timestamp-uri perfect periodice, $D_i = 0$ pentru toate $i > 1$. Aceasta este observația crucială: dacă timestamp-urile sunt regulate, **delta-of-delta este zero**.

Exemplu 3.1. Să considerăm timestamp-urile: $1000, 1060, 1120, 1185, 1245$.

i	t_i	$\delta_i = t_i - t_{i-1}$	$D_i = \delta_i - \delta_{i-1}$
0	1000	—	—
1	1060	60	—
2	1120	60	0
3	1185	65	+5
4	1245	60	-5

Observăm că $D_2 = 0$ (periodicitate perfectă), iar D_3 și D_4 sunt valori mici (± 5) care pot fi stocate eficient.

3.2.4 Schema de codare variabilă

Gorilla utilizează o schemă de codare cu lungime variabilă pentru valorile delta-of-delta, optimizată pentru distribuția observată în date reale:

Condiție	**Prefix**	**Valoare**	**Total biți**
$D = 0$	`0`	—	1
$D \in [-64, 63]$	`10`	7 biți signed	9
$D \in [-256, 255]$	`110`	9 biți signed	12
$D \in [-2048, 2047]$	`1110`	12 biți signed	16
Altfel	`1111`	32 biți signed	36

Tabela 3.1: Schema de codare pentru delta-of-delta

3.2.5 Reprezentarea în complement față de 2

Valorile delta-of-delta pot fi negative, deci trebuie reprezentate ca numere cu semn. Gorilla utilizează reprezentarea în **two's complement**.

Pentru un număr x reprezentat pe n biți:

- Dacă $x \geq 0$: reprezentarea este x în binar;
- Dacă $x < 0$: reprezentarea este $2^n + x$.

Intervalul reprezentabil pe n biți este $[-2^{n-1}, 2^{n-1} - 1]$.

Exemplu 3.2. Pentru $n = 7$ biți (intervalul $[-64, 63]$):

- $x = 5$: reprezentare = 0000101_2
- $x = -5$: reprezentare = $2^7 + (-5) = 128 - 5 = 123 = 1111011_2$

3.2.6 Algoritmul de codare

Este prezentat în cadrul **Algorithm 1**.

3.2.7 Algoritmul de decodare

Decodarea citește biții de control pentru a determina lungimea valorii. Vezi **Algorithm 2**.

3.2.8 Eficiența compresiei timestamp-urilor

În datele reale de la Facebook, distribuția valorilor delta-of-delta este puternic concentrată în jurul lui zero:

- **96.39%** dintre timestamp-uri au $D = 0$ (comprimate la 1 bit);
- **3.35%** au $D \in [-64, 63]$ (9 biți);
- **0.19%** au $D \in [-256, 255]$ (12 biți);

Algorithm 1 Codare timestamp cu delta-of-delta

Require: Timestamp t_n, timestamp anterior t_{n-1}, delta anterior δ_{n-1}
Ensure: Biți scriși în fluxul de ieșire

$\delta_n \leftarrow t_n - t_{n-1}$
$D \leftarrow \delta_n - \delta_{n-1}$
if $D = 0$ **then**
    write `0` ▷ 1 bit
else if $-64 \leq D \leq 63$ **then**
    write `10` ▷ 2 biți prefix
    write D pe 7 biți în complement față de 2
else if $-256 \leq D \leq 255$ **then**
    write `110` ▷ 3 biți prefix
    write D pe 9 biți în complement față de 2
else if $-2048 \leq D \leq 2047$ **then**
    write `1110` ▷ 4 biți prefix
    write D pe 12 biți în complement față de 2
else
    write `1111` ▷ 4 biți prefix
    write D pe 32 biți în complement față de 2
end if
return δ_n ▷ pentru următoarea iterație

- Restul utilizează 16 sau 36 de biți.

Media ponderată rezultă în aproximativ **1.5 biți per timestamp**, comparativ cu 64 de biți fără compresie.

3.3 Compresia Valorilor: XOR Encoding

3.3.1 Provocarea compresiei floating-point

Numerele în virgulă mobilă (floating-point) sunt reprezentate conform standardului IEEE 754. Un număr *double* (64 biți) are structura:

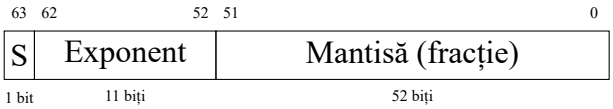

Figura 3.2: Structura IEEE 754 double-precision (64 biți)

Valoarea reprezentată este:

$$v = (-1)^S \times 2^{E-1023} \times (1 + M/2^{52}) \tag{3.4}$$

unde S este bitul de semn, E este exponentul (11 biți), și M este mantisa (52 biți).

Algorithm 2 Decodare timestamp

Require: Flux de biți, timestamp anterior t_{n-1}, delta anterior δ_{n-1}
Ensure: Timestamp decodat t_n

$b_1 \leftarrow$ **read** 1 bit
if $b_1 = 0$ **then**
    $D \leftarrow 0$
else
    $b_2 \leftarrow$ **read** 1 bit
    if $b_2 = 0$ **then**
        $D \leftarrow$ **read** 7 biți signed
    else
        $b_3 \leftarrow$ **read** 1 bit
        if $b_3 = 0$ **then**
            $D \leftarrow$ **read** 9 biți signed
        else
            $b_4 \leftarrow$ **read** 1 bit
            if $b_4 = 0$ **then**
                $D \leftarrow$ **read** 12 biți signed
            else
                $D \leftarrow$ **read** 32 biți signed
            end if
        end if
    end if
end if
$\delta_n \leftarrow \delta_{n-1} + D$
$t_n \leftarrow t_{n-1} + \delta_n$
return t_n

Problema: Chiar și pentru valori foarte apropiate, reprezentările binare pot diferi semnificativ la nivelul mantisei. Delta encoding simplu nu funcționează eficient.

3.3.2 Observația cheie: XOR pentru valori similare

Gorilla exploatează o proprietate importantă: dacă două valori floating-point sunt *apropiate numeric*, reprezentările lor binare tind să aibă mulți biți identici, în special în exponent și în biții superiori ai mantisei.

Operația **XOR** (exclusive OR) între două reprezentări binare produce:

- `0` pentru biții identici;
- `1` pentru biții diferiți.

Pentru valori apropiate, XOR-ul va avea mulți `0` consecutivi la început (*leading zeros*) și la sfârșit (*trailing zeros*).

Exemplu 3.3. Fie $v_1 = 24.0$ și $v_2 = 25.0$:

$$v_1 = \texttt{0x4038000000000000}$$
$$v_2 = \texttt{0x4039000000000000}$$
$$v_1 \oplus v_2 = \texttt{0x0001000000000000}$$

XOR-ul are 15 leading zeros și 48 trailing zeros, lăsând doar 1 bit semnificativ!

3.3.3 Schema de codare XOR

Gorilla utilizează următoarea schemă pentru codarea valorilor:

Prima valoare

Prima valoare din bloc se stochează integral pe 64 de biți (necomprimată).

Valori ulterioare

Pentru fiecare valoare v_n după prima:

1. Calculăm $X = v_n \oplus v_{n-1}$ (XOR cu valoarea anterioară);
2. **Dacă** $X = 0$ (valori identice):
   - Scriem un singur bit `0`;
3. **Dacă** $X \neq 0$:
   - Scriem bitul de control `1`;
   - Calculăm *leading zeros* (L) și *trailing zeros* (T);
   - Numărul de biți semnificativi este: $M = 64 - L - T$;
4. **Dacă putem refolosi fereastra anterioară** ($L \geq L_{prev}$ și $T \geq T_{prev}$):
   - Scriem bitul de control `0`;
   - Scriem doar biții semnificativi (M_{prev} biți);
5. **Altfel** (definim o fereastră nouă):
   - Scriem bitul de control `1`;
   - Scriem L pe 5 biți (permite valori 0-31);
   - Scriem $M - 1$ pe 6 biți (permite valori 1-64 codate ca 0-63);
   - Scriem cei M biți semnificativi din XOR.

Caz	**Format**	**Biți**
XOR = 0	`0`	1
Refolosim fereastra	`10` + meaningful bits	2 + M_{prev}
Fereastră nouă	`11` + 5 biți + 6 biți + meaningful bits	13 + M

Tabela 3.2: Schema de codare XOR pentru valori

3.3.4 Calculul leading și trailing zeros

Fie X rezultatul XOR reprezentat pe 64 de biți. Definim:

$$L = \max\{i : \text{biții } 63, 62, \ldots, 64-i \text{ sunt toți } 0\} \tag{3.5}$$

$$T = \max\{j : \text{biții } 0, 1, \ldots, j-1 \text{ sunt toți } 0\} \tag{3.6}$$

$$M = 64 - L - T \quad \text{(biții semnificativi)} \tag{3.7}$$

Exemplu 3.4. Pentru $X = \texttt{0x0001000000000000}$:

- Reprezentare binară: `0000...0001 0000...0000` (15 zeros, apoi 1, apoi 48 zeros)
- $L = 15$ (leading zeros)
- $T = 48$ (trailing zeros)
- $M = 64 - 15 - 48 = 1$ (un singur bit semnificativ)

3.3.5 Algoritmul de codare pentru valori

Algorithm 3 Codare valoare cu XOR

Require: Valoare v_n, valoare anterioară v_{n-1} (ca biți), L_{prev}, T_{prev}
Ensure: Biți scriși în fluxul de ieșire

$X \leftarrow v_n \oplus v_{n-1}$ ▷ XOR între reprezentări pe 64 biți
if $X = 0$ **then**
  write `0` ▷ 1 bit — valori identice
else
  write `1` ▷ bit de control
  $L \leftarrow$ CountLeadingZeros(X)
  $T \leftarrow$ CountTrailingZeros(X)
  $M \leftarrow 64 - L - T$
  if $L \geq L_{prev}$ **and** $T \geq T_{prev}$ **then**
    write `0` ▷ refolosim fereastra
    $M_{use} \leftarrow 64 - L_{prev} - T_{prev}$
    write biții semnificativi din X (M_{use} biți)
  else
    write `1` ▷ fereastră nouă
    write $\min(L, 31)$ pe 5 biți
    write $(M - 1)$ pe 6 biți
    write biții semnificativi din X (M biți)
    $L_{prev} \leftarrow L$; $T_{prev} \leftarrow T$
  end if
end if

3.3.6 Eficiența compresiei valorilor

În datele Facebook:

- **59.06%** dintre valori sunt identice cu precedenta (1 bit);
- **28.30%** refolosesc fereastra anterioară (27 biți în medie);
- **12.64%** necesită fereastră nouă (40 biți în medie).

Media rezultă în aproximativ **12-15 biți per valoare**, comparativ cu 64 de biți fără compresie.

3.4 Compresia combinată

Rata totală de compresie pentru o pereche (timestamp, valoare) este:

$$R = \frac{128 \text{ biți}}{b_{ts} + b_{val}} \tag{3.8}$$

unde b_{ts} și b_{val} sunt biții necesari pentru timestamp și valoare.

În practică, Gorilla obține în medie **1.37 bytes per punct** (11 biți), comparativ cu 16 bytes necomprimat, reprezentând o rată de compresie de aproximativ **12x**.

3.5 Analiza complexității

3.5.1 Complexitate temporală

Codare (per punct): $O(1)$

- Calcul delta/XOR: $O(1)$
- Scriere biți: $O(k)$ unde k este numărul de biți (constant, maxim 100)

Decodare (per punct): $O(1)$

- Citire biți de control: $O(1)$
- Reconstrucție valoare: $O(1)$

3.5.2 Complexitate spațială

Stare per encoder: $O(1)$

- Timestamp anterior: 8 bytes
- Delta anterior: 8 bytes
- Valoare anterioară: 8 bytes
- Leading/trailing zeros anteriori: 2 bytes

Buffer de ieșire: $O(n)$ unde n este numărul de puncte, dar cu factor constant mic (≈ 1.37 bytes/punct în loc de 16).

3.6 Extindere pentru serii multivariate

O serie **multivariată** conține mai multe valori per timestamp:

$$(t_i, v_i^{(1)}, v_i^{(2)}, \ldots, v_i^{(k)}) \tag{3.9}$$

3.6.1 Strategia de compresie

Abordarea naivă ar fi să tratăm fiecare variabilă ca o serie separată, dar aceasta ar duplica timestamp-urile de k ori.

Gorilla pentru serii multivariate utilizează:

1. **Un singur stream de timestamp-uri** (delta-of-delta);
2. **Câte un stream separat pentru fiecare variabilă** (XOR encoding).

Toate stream-urile scriu în același buffer de biți, dar fiecare variabilă își menține propriul context XOR (valoare anterioară, leading/trailing zeros).

Capitolul 4

Implementare

Acest capitol prezintă arhitectura și clasele implementării algoritmului Gorilla în Python. Detaliile teoretice ale algoritmului (schema de codare delta-of-delta, XOR encoding, reprezentarea în complement față de 2) au fost prezentate în Capitolul 3. Aici ne concentrăm pe structurile de date, relațiile dintre componente și pe o variantă îmbunătățită a algoritmului original.

4.1 Arhitectura sistemului

4.1.1 Diagrama dependențelor

Sistemul este organizat pe trei niveluri ierarhice, prezentate în Figura 4.1. La bază se află operațiile pe biți, urmate de encoderele și decoderele specializate, iar la vârf se află modulele de stocare.

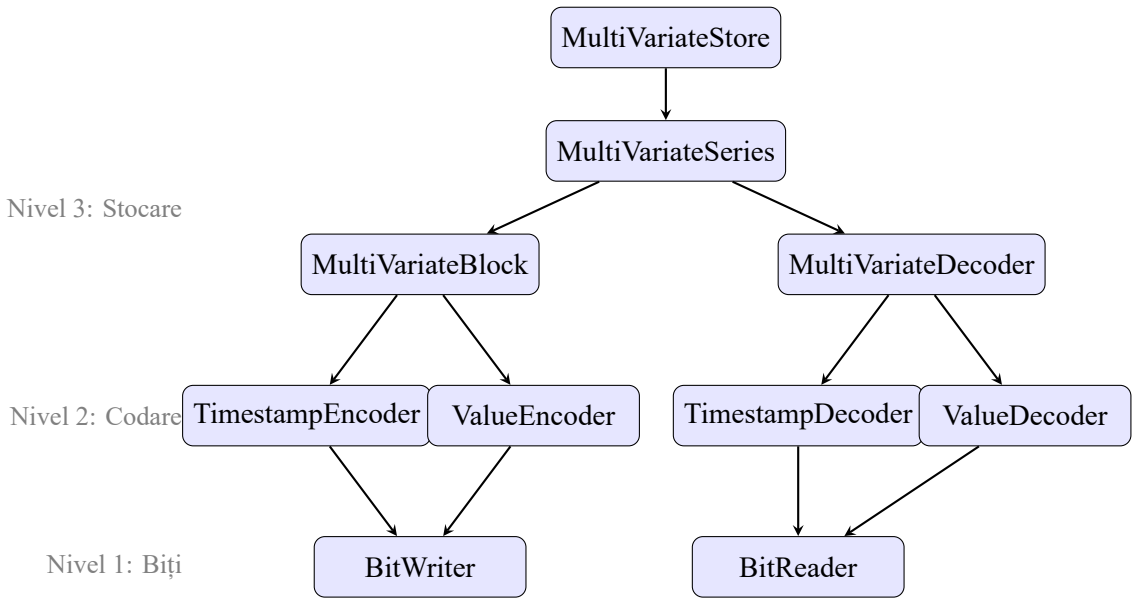

Figura 4.1: Ierarhia dependențelor dintre clase

4.1.2 Principii de design

Toate clasele utilizează directiva `__slots__` pentru optimizare. Aceasta elimină dicționarul implicit al obiectelor Python (`__dict__`), rezultând în:

- Reducerea amprentei de memorie per obiect
- Acces mai rapid la atribute (fără lookup în dicționar)
- Protecție la erori de tip (atributele nedeclarate generează excepții)

4.2 Nivelul 1: Operații pe biți

4.2.1 Clasa BitWriter

Rol: Permite scrierea de secvențe de biți de lungime arbitrară într-un buffer de bytes. Algoritmul Gorilla necesită scrierea de câmpuri de 1, 5, 7, 9, 12 sau 32 de biți care nu se aliniază la granița de byte.

Structura de date internă:

Atribut	Tip	Descriere
`_buf`	`bytearray`	Buffer cu bytes compleți (finalizați)
`_cur`	`int`	Byte parțial în construcție (0–255)
`_nbits`	`int`	Numărul de biți scriși în `_cur` (0–7)

Modelul de acumulare: Biții se scriu de la stânga la dreapta (MSB-first). Când `_nbits` atinge 8, byte-ul complet este mutat în buffer și se resetează `_cur`.

Metodă	Descriere
`write_bit(bit)`	Scrie un singur bit (0 sau 1)
`write_bits(x, n)`	Scrie n biți din valoarea x
`write_signed(x, bits)`	Scrie x ca număr cu semn (complement față de 2)
`write_u64(x)` / `write_i64(x)`	Scrie întreg pe 64 biți (aliniat la byte)
`to_bytes()`	Finalizează și returnează `bytes`

4.2.2 Clasa BitReader

Rol: Clasă complementară pentru citirea bit cu bit dintr-un flux de bytes. Reconstruiește valorile scrise cu BitWriter.

Structura de date internă:

Atribut	Tip	Descriere
`_data`	`bytes`	Sursa de date (imutabilă)
`_byte_pos`	`int`	Indexul byte-ului curent
`_bit_pos`	`int`	Poziția bitului în byte (0–7)

Metodă	Descriere
`read_bit()`	Citește un singur bit
`read_bits(n)`	Citește n biți ca întreg unsigned
`read_signed(bits)`	Citește număr cu semn (complement față de 2)
`read_u64()` / `read_i64()`	Citește întreg pe 64 biți
`bits_remaining`	Proprietate: biți rămași în flux

4.3 Nivelul 2: Encodere și decodere

Schema de codare delta-of-delta pentru timestamp-uri și XOR encoding pentru valori au fost detaliate în Capitolul 3. Acum prezentăm structura internă a claselor care implementează aceste scheme.

4.3.1 Clasa TimestampEncoder

Rol: Comprimă timestamp-uri folosind codarea delta-of-delta.

Atribut	Semnificație
`_writer`	Referință la BitWriter pentru scriere
`_prev_timestamp`	Ultimul timestamp procesat (t_{i-1})
`_prev_delta`	Ultimul delta calculat (δ_{i-1})
`_count`	Numărul de timestamp-uri adăugate

Logica de codare:

- Primul timestamp: se scrie complet pe 64 de biți
- Al doilea timestamp: se scrie $\delta_1 = t_1 - t_0$ pe 64 de biți
- Timestamp-urile următoare: se codează delta-of-delta conform schemei din Tabelul 3.1

4.3.2 Clasa TimestampDecoder

Rol: Reconstruiește timestamp-urile din fluxul comprimat. Inversează operațiile TimestampEncoder.

Starea internă: Identică cu TimestampEncoder, dar folosește BitReader în loc de BitWriter.

4.3.3 Clasa ValueEncoder

Rol: Comprimă valori `float64` folosind codarea XOR.

Atribut	Semnificație
`_writer`	Referință la BitWriter
`_prev_value_bits`	Reprezentarea pe biți a valorii anterioare
`_prev_leading`	Numărul de zerouri din față (fereastra anterioară)
`_prev_trailing`	Numărul de zerouri din coadă (fereastra anterioară)
`_count`	Numărul de valori procesate

4.3.4 Clasa ValueDecoder

Rol: Reconstruiește valorile float din fluxul comprimat XOR.

4.4 Nivelul 3: Stocare multivariată

4.4.1 Clasa MultiVariateBlock

Rol: Gestionează un bloc de date pentru serii cu multiple variabile. Toate variabilele împart același stream de timestamp-uri, evitând duplicarea.

Structura de date:

Atribut	Descriere
`_writer`	BitWriter comun pentru toate stream-urile
`_ts_encoder`	Un singur TimestampEncoder
`_val_encoders`	Dicționar: nume variabilă $\rightarrow$ ValueEncoder
`_var_names`	Lista ordonată a numelor variabilelor
`_count`	Numărul de puncte din bloc
`_closed`	Flag: blocul este finalizat
`_start_timestamp`	Timestamp-ul de start al blocului

Principiul cheie: Un singur timestamp pentru N variabile. Ordinea variabilelor trebuie să fie **deterministă** la codare și decodare.

Metodă	Descriere
`add(ts, values)`	Adaugă punct (algoritmul Gorilla standard)
`add_verification(ts, values)`	Adaugă punct (varianta îmbunătățită)
`seal()`	Finalizează blocul, returnează bytes comprimați

4.4.2 Clasa MultiVariateDecoder

Rol: Citește și reconstruiește punctele multivariate dintr-un bloc comprimat.

Structură: Un TimestampDecoder + un dicționar de ValueDecodere (unul per variabilă).

4.4.3 Clasa MultiVariateSeries

Rol: Gestionează o serie temporală completă, organizată în blocuri de durată fixă (implicit 2 ore).

Structura de date:

Atribut	Descriere
`_var_names`	Lista numelor variabilelor
`_block_duration`	Durata unui bloc în milisecunde
`_open_block`	Blocul curent (în care se scrie)
`_closed_blocks`	Listă: (start_ts, count, bytes)

Interfață:

Metodă	Descriere
`insert(ts, values)`	Inserează punct (gestionează blocuri automat)
`flush()`	Închide blocul curent
`query(t_start, t_end)`	Interogare pe interval temporal
`get_compression_stats()`	Returnează statistici de compresie

4.4.4 Clasa MultiVariateStore

Rol: Container pentru mai multe serii temporale multivariate, identificate prin chei unice.

Structură: Dicționar `serie_key` → `MultiVariateSeries`.

4.5 Varianta îmbunătățită: Optimizarea ferestrei XOR

În această secțiune prezentăm o modificare a algoritmului Gorilla original, implementată în metodele `add_value_verification` și `add_verification`.

4.5.1 Problema algoritmului original

În algoritmul Gorilla standard, decizia de a refolosi fereastra anterioară se bazează exclusiv pe condiția:

$$L_{curent} \geq L_{anterior} \quad \text{și} \quad T_{curent} \geq T_{anterior} \tag{4.1}$$

unde L = leading zeros și T = trailing zeros.

Problema: Această condiție poate duce la ineficiențe. Dacă fereastra anterioară are $M_{anterior} = 64 - L_{anterior} - T_{anterior}$ biți semnificativi, iar valoarea curentă ar necesita doar $M_{curent} = 64 - L_{curent} - T_{curent}$ biți, algoritmul original va scrie $M_{anterior}$ biți chiar dacă $M_{curent} \ll M_{anterior}$.

Exemplu 4.1. Fie fereastra anterioară cu $L_{ant} = 10$, $T_{ant} = 20$, deci $M_{ant} = 34$ biți.

Valoarea curentă are $L_{cur} = 15$, $T_{cur} = 40$, deci $M_{cur} = 9$ biți.

Condiția originală ($15 \geq 10$ și $40 \geq 20$) este satisfăcută, deci algoritmul va refolosi fereastra și va scrie 34 de biți în loc de 9.

Costul refolosirii: 2 (prefix `10`) + 34 (biți) = 36 biți

Costul ferestrei noi: 2 (prefix `11`) + 5 (leading) + 6 (lungime) + 9 (biți) = 22 biți

Diferența: 14 biți irosiți per valoare!

4.5.2 Soluția propusă

Varianta îmbunătățită adaugă o condiție suplimentară: se verifică dacă fereastra anterioară este **semnificativ** mai mare decât fereastra curentă. Crearea unei ferestre noi costă exact 11 biți (5 pentru leading zeros + 6 pentru lungime). Prin urmare, merită să creăm o fereastră nouă doar dacă economia de biți depășește acest overhead:

$$M_{anterior} - M_{curent} > 11 \tag{4.2}$$

Condiția completă pentru refolosirea ferestrei:

$$L_{cur} \geq L_{ant} \quad \wedge \quad T_{cur} \geq T_{ant} \quad \wedge \quad (M_{ant} - M_{cur} \leq 11) \tag{4.3}$$

4.5.3 Implementarea

În fișierul `value_compression.py`, metoda `add_value_verification` implementează această logică:

```
# Varianta standard (add_value)
if (self._prev_leading != 255 and
    leading >= self._prev_leading and
    trailing >= self._prev_trailing):
    # Refolosim fereastra

# Varianta imbunatatita (add_value_verification)
if (self._prev_leading != 255 and
    leading >= self._prev_leading and
    trailing >= self._prev_trailing and
    not ((64 - self._prev_trailing - self._prev_leading)
         - (64 - trailing - leading) > 11)):
    # Refolosim fereastra doar daca diferenta <= 11 biti
```

Listing 4.1: Condiția îmbunătățită pentru refolosirea ferestrei

Condiția `not (M_ant - M_cur > 11)` se traduce în: "refolosește fereastra doar dacă economia potențială nu depășește overhead-ul de 11 biți".

4.5.4 Analiza trade-off-ului

Decizia optimă se bazează pe compararea costurilor:

Opțiune	**Cost (biți)**
Refolosire fereastră	$2 + M_{anterior}$
Fereastră nouă	$2 + 11 + M_{curent}$

Fereastra nouă este mai eficientă când:

$$2 + 11 + M_{curent} < 2 + M_{anterior} \implies M_{anterior} - M_{curent} > 11 \tag{4.4}$$

Această condiție este **matematic optimă**: se creează fereastră nouă exact atunci când este benefic.

4.6 Fluxul de date

4.6.1 Codare (compresie)

1. Aplicația apelează `MultiVariateSeries.insert(ts, values)`
2. Seria verifică dacă e nevoie de bloc nou și apelează `MultiVariateBlock.add()`
3. Blocul apelează `TimestampEncoder.add_timestamp(ts)`
4. Encoder-ul timestamp calculează delta-of-delta și scrie în BitWriter
5. Pentru fiecare variabilă, blocul apelează `ValueEncoder.add_value(v)`
6. Fiecare ValueEncoder calculează XOR și scrie în același BitWriter
7. La `flush()`, BitWriter returnează bytes-ii comprimați

4.6.2 Decodare (decompresie)

1. Se creează `MultiVariateDecoder` cu bytes comprimați
2. Decoder-ul inițializează BitReader + TimestampDecoder + ValueDecoders
3. La fiecare `read_point()`:
    - TimestampDecoder citește și reconstruiește timestamp-ul
    - Fiecare ValueDecoder citește și reconstruiește valoarea sa
4. Se returnează tuplu (timestamp, dicționar_valori)

Capitolul 5

Experimente și Rezultate

Acest capitol prezintă evaluarea experimentală a implementării algoritmului Gorilla. Sunt descrise seturile de date utilizate, metodologia de testare și rezultatele obținute. Demonstrăm mai întâi eficiența compresiei Gorilla față de stocarea necomprimată, apoi prezentăm îmbunătățirea adusă de varianta optimizată propusă în această lucrare.

5.1 Seturi de date

Pentru validarea implementării au fost utilizate trei seturi de date cu caracteristici diferite: două serii univariate și una multivariată.

5.1.1 CPU Load Average (serie univariată)

Primul set de date conține metrici de încărcare CPU colectate de pe un sistem Linux.

Caracteristică	**Valoare**
Număr de puncte	480
Perioadă de colectare	$\sim$16 ore
Interval mediu	$\sim$120 secunde
Tip date	Univariat (load average)
Interval valori	0.57 – 2.55

Tabela 5.1: Caracteristicile setului de date CPU Load

Load average reprezintă numărul mediu de procese în coadă de execuție pe o fereastră de timp. Valorile variază ușor în timp, cu schimbări graduale caracteristice sarcinilor tipice de server.

5.1.2 Twitter Volume UPS (serie univariată)

Al doilea set de date provine din arhiva Numenta Anomaly Benchmark și conține volumul de tweet-uri care menționează compania UPS.

Caracteristică	Valoare
Număr de puncte	15,866
Perioadă de colectare	∼55 zile
Interval mediu	5 minute (constant)
Tip date	Univariat (volum tweet-uri)
Interval valori	0 – 2,523

Tabela 5.2: Caracteristicile setului de date Twitter Volume

Acest set de date este deosebit de relevant deoarece:

- **Timestamp-uri perfect periodice**: Interval exact de 5 minute, ideal pentru compresia delta-of-delta
- **Valori întregi**: Contoare de tweet-uri, care produc pattern-uri XOR diferite față de valorile continue
- **Dimensiune medie**: Cu 15,866 puncte, permite evaluarea scalabilității

5.1.3 Room Climate Dataset (serie multivariată)

Al treilea set de date provine din Room Climate Dataset, o colecție de măsurători de la senzori de mediu interior [**roomclimate**].

Caracteristică	Valoare
Număr de puncte	68,229
Perioadă de colectare	∼16 zile
Interval mediu	∼4 secunde
Număr variabile	8
Node ID filtrat	NID = 1

Tabela 5.3: Caracteristicile setului de date Room Climate

Cele 8 variabile măsurate sunt: temperatură, umiditate relativă, lumină 1, lumină 2, ocupare, activitate, stare ușă și stare fereastră.

5.2 Metodologia experimentală

5.2.1 Metrici de evaluare

Au fost calculate următoarele metrici:

1. **Dimensiune necomprimată**: $n \times (8 + 8k)$ bytes, unde n este numărul de puncte și k numărul de variabile. Aceasta corespunde stocării naive: 8 bytes pentru timestamp (int64) + 8 bytes per valoare (float64).
2. **Dimensiune comprimată**: numărul de bytes după aplicarea algoritmului Gorilla.
3. **Rata de compresie**:
$$R = \frac{\text{Dimensiune necomprimată}}{\text{Dimensiune comprimată}} \tag{5.1}$$
4. **Îmbunătățire procentuală** (economie de spațiu):
$$E = \left(1 - \frac{\text{Dimensiune comprimată}}{\text{Dimensiune originală}}\right) \times 100\% \tag{5.2}$$

5.2.2 Configurația testelor

- Blocuri de 2 ore (7.200.000 ms), conform configurației standard Gorilla
- Validare round-trip pentru fiecare test (compresie → decompresie → verificare bit-perfect)

5.3 Rezultate: Tabel comparativ principal

5.3.1 Comparație globală

Dataset	Necmp.	Gorilla Std	Gorilla Opt	Îmbun. Necmp→Std	Îmbun. Std→Opt
CPU Load	7,680 B	3,485 B	3,236 B	54.6%	7.14%
Twitter Vol.	253,856 B	38,520 B	38,258 B	84.8%	0.68%
Room Climate	4,912,488 B	1,190,271 B	1,169,018 B	75.8%	1.79%
TOTAL	5,174,024 B	1,232,276 B	1,210,512 B	**76.2%**	**1.77%**

Tabela 5.4: Comparație completă: Necomprimat vs Gorilla Standard vs Gorilla Optimizat

Interpretare:
- Coloana **Îmbun. Necmp→Std**: Cât de mult reduce Gorilla Standard față de stocarea necomprimată
- Coloana **Îmbun. Std→Opt**: Cât de mult îmbunătățește varianta noastră optimizată față de Gorilla Standard

5.4 Rezultate detaliate: CPU Load

Analiza: CPU Load beneficiază cel mai mult de optimizare (**7.14%**). Valorile load average variază gradual, producând multe situații în care fereastra anterioară era semnificativ mai mare decât necesarul. Economie suplimentară: **249 bytes**.

Metrică	Necmp.	Gorilla Std	Gorilla Opt	Îmbunătățiri
Dimensiune	7,680 B	3,485 B	3,236 B	–
Bytes per punct	16.00	7.26	6.74	–
Biți per punct	128.00	58.10	53.95	–
Rata compresie	1.00x	2.20x	2.37x	–
Îmbunătățire Necomprimat → Gorilla Standard				**54.6%**
Îmbunătățire Gorilla Standard → Gorilla Optimizat				**7.14%**

Tabela 5.5: CPU Load: comparație detaliată cu îmbunătățiri

5.5 Rezultate detaliate: Twitter Volume

Metrică	Necmp.	Gorilla Std	Gorilla Opt	Îmbunătățiri
Dimensiune	253,856 B	38,520 B	38,258 B	–
Bytes per punct	16.00	2.43	2.41	–
Biți per punct	128.00	19.42	19.29	–
Rata compresie	1.00x	6.59x	6.64x	–
Îmbunătățire Necomprimat → Gorilla Standard				**84.8%**
Îmbunătățire Gorilla Standard → Gorilla Optimizat				**0.68%**

Tabela 5.6: Twitter Volume: comparație detaliată cu îmbunătățiri

Analiza: Acest set demonstrează cele mai bune rezultate pentru Gorilla (**84.8%** reducere) datorită timestamp-urilor perfect periodice. Îmbunătățirea optimizării este mai modestă (0.68%) deoarece valorile întregi produc XOR-uri cu pattern-uri mai previzibile. Economie suplimentară: **262 bytes**.

5.6 Rezultate detaliate: Room Climate

Metrică	Necmp.	Gorilla Std	Gorilla Opt	Îmbunătățiri
Dimensiune	4,912,488 B	1,190,271 B	1,169,018 B	–
Bytes per punct	72.00	17.45	17.13	–
Biți per punct	576.00	139.55	137.06	–
Rata compresie	1.00x	4.13x	4.20x	–
Îmbunătățire Necomprimat → Gorilla Standard				**75.8%**
Îmbunătățire Gorilla Standard → Gorilla Optimizat				**1.79%**

Tabela 5.7: Room Climate: comparație detaliată cu îmbunătățiri

Analiza: Cu 8 variabile și 68,229 puncte, acest set demonstrează scalabilitatea atât a algoritmului Gorilla (economie de **3.7 MB**), cât și a optimizării noastre (economie suplimentară de

21,253 bytes ≈ 20.7 KB).

5.7 Sumar rate de compresie

Dataset	Rata Std	Rata Opt	Îmbun. Necmp→Std	Îmbun. Std→Opt
CPU Load	2.20x	2.37x	54.6%	7.14%
Twitter Volume	6.59x	6.64x	84.8%	0.68%
Room Climate	4.13x	4.20x	75.8%	1.79%
Media	4.31x	4.40x	**71.7%**	**3.20%**

Tabela 5.8: Sumar rate de compresie și îmbunătățiri

5.8 Varianta optimizată: Recapitulare

5.8.1 Condiția standard vs optimizată

Varianta standard Gorilla refolosește fereastra XOR anterioară dacă:

$$L_{cur} \geq L_{ant} \quad \wedge \quad T_{cur} \geq T_{ant} \tag{5.3}$$

Varianta noastră adaugă o condiție suplimentară:

$$L_{cur} \geq L_{ant} \quad \wedge \quad T_{cur} \geq T_{ant} \quad \wedge \quad (M_{ant} - M_{cur} \leq 11) \tag{5.4}$$

unde $M = 64 - L - T$ este numărul de biți semnificativi. Această condiție evită refolosirea ineficientă a unei ferestre prea mari.

5.8.2 Exemplu numeric

Fie un scenariu concret:

- Valoarea anterioară: fereastră de 45 biți semnificativi ($L = 10, T = 9$)
- Valoarea curentă: necesită doar 8 biți semnificativi ($L = 28, T = 28$)

Variantă	Decizie	Calcul	Cost
Standard	Refolosire	$2 + 45$	47 biți
Optimizat	Fereastră nouă	$2 + 5 + 6 + 8$	21 biți
Economie per punct			**26 biți**

Tabela 5.9: Comparație cost pentru un singur punct

5.9 Comparație cu articolul original Gorilla

Metrică	Facebook	Impl. Std	Impl. Opt
Bytes/punct	1.37	2.4–7.3	2.4–6.7
% timestamps D=0	96.4%	85%–99%	85%–99%
% valori XOR=0	59.1%	∼45%	∼45%
Rata compresie	12x	2.2–6.6x	2.4–6.6x

Tabela 5.10: Comparație cu rezultatele raportate în articolul Gorilla

Diferențele se explică prin caracteristicile datelor: Facebook folosește metrici de monitorizare foarte regulate, în timp ce seturile noastre au mai multă variabilitate.

5.10 Validarea corectitudinii

Toate testele au trecut validarea round-trip, confirmând caracterul *lossless*:

```
# Comprima
for ts, vals in original_data:
    series.insert(ts, vals)
series.flush()

# Decomprima
decoder = MultiVariateDecoder(compressed_data, var_names)
decoded = [decoder.read_point() for _ in range(count)]

# Verifica (bit-perfect)
for (ts_orig, vals_orig), (ts_dec, vals_dec) in zip(original,
    decoded):
    assert ts_orig == ts_dec
    for var in var_names:
        assert vals_orig[var] == vals_dec[var]
```

Listing 5.1: Verificare round-trip

5.11 Concluzii

Experimentele demonstrează:

1. **Eficiența Gorilla**: Algoritmul reduce dimensiunea datelor cu **55–85%** față de stocarea necomprimată, confirmând valoarea sa pentru serii temporale.

Concluzie	Îmbun. Necmp→Std	Îmbun. Std→Opt
CPU Load	54.6%	7.14%
Twitter Volume	84.8%	0.68%
Room Climate	75.8%	1.79%
Total/Medie	**76.2%**	**1.77%**

Tabela 5.11: Sumar final al îmbunătățirilor

2. **Optimizarea funcționează**: Pe toate cele 3 seturi de date, varianta optimizată produce fișiere mai mici decât cea standard, cu îmbunătățiri între **0.68% și 7.14%**.
3. **Fără compromisuri**: Nu există niciun caz în care varianta standard ar fi mai bună — optimizarea este **universal benefică**.
4. **Scalabilitate**: Cu cât mai multe date, cu atât economia absolută este mai mare (21+ KB pentru Room Climate).

Recomandare: Varianta optimizată ar trebui folosită întotdeauna în locul celei standard, fiind superioară din punct de vedere matematic și confirmat experimental.

5.12 Anexă: Vizualizări grafice

Această secțiune prezintă vizualizări grafice ale seturilor de date și rezultatelor experimentale.

5.12.1 Seria CPU Load

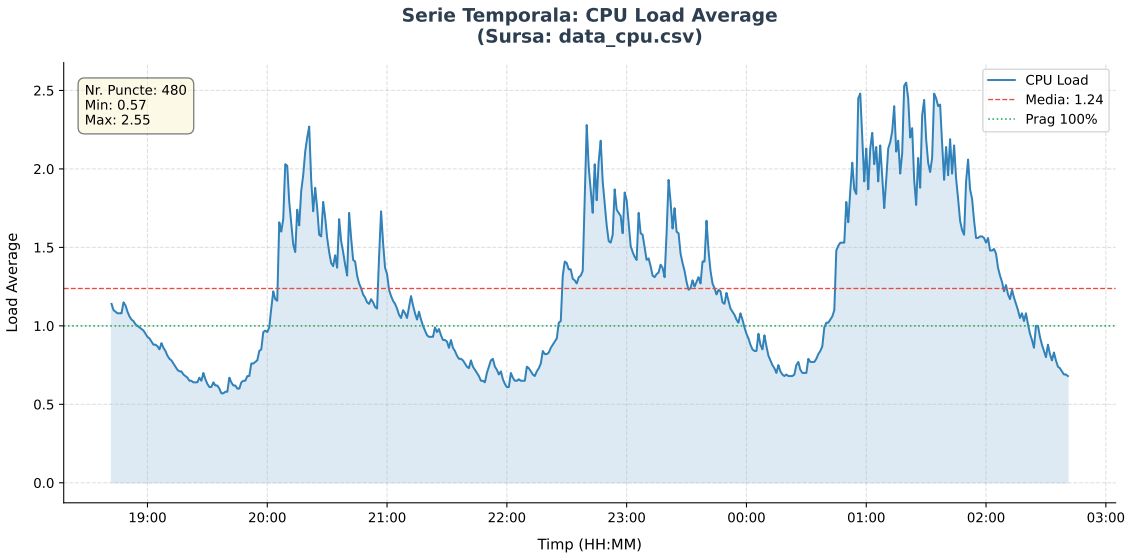

Figura 5.1: Vizualizarea seriei temporale CPU Load Average. Se observă variația graduală a valorilor load average pe parcursul perioadei de colectare.

5.12.2 Seria Room Climate

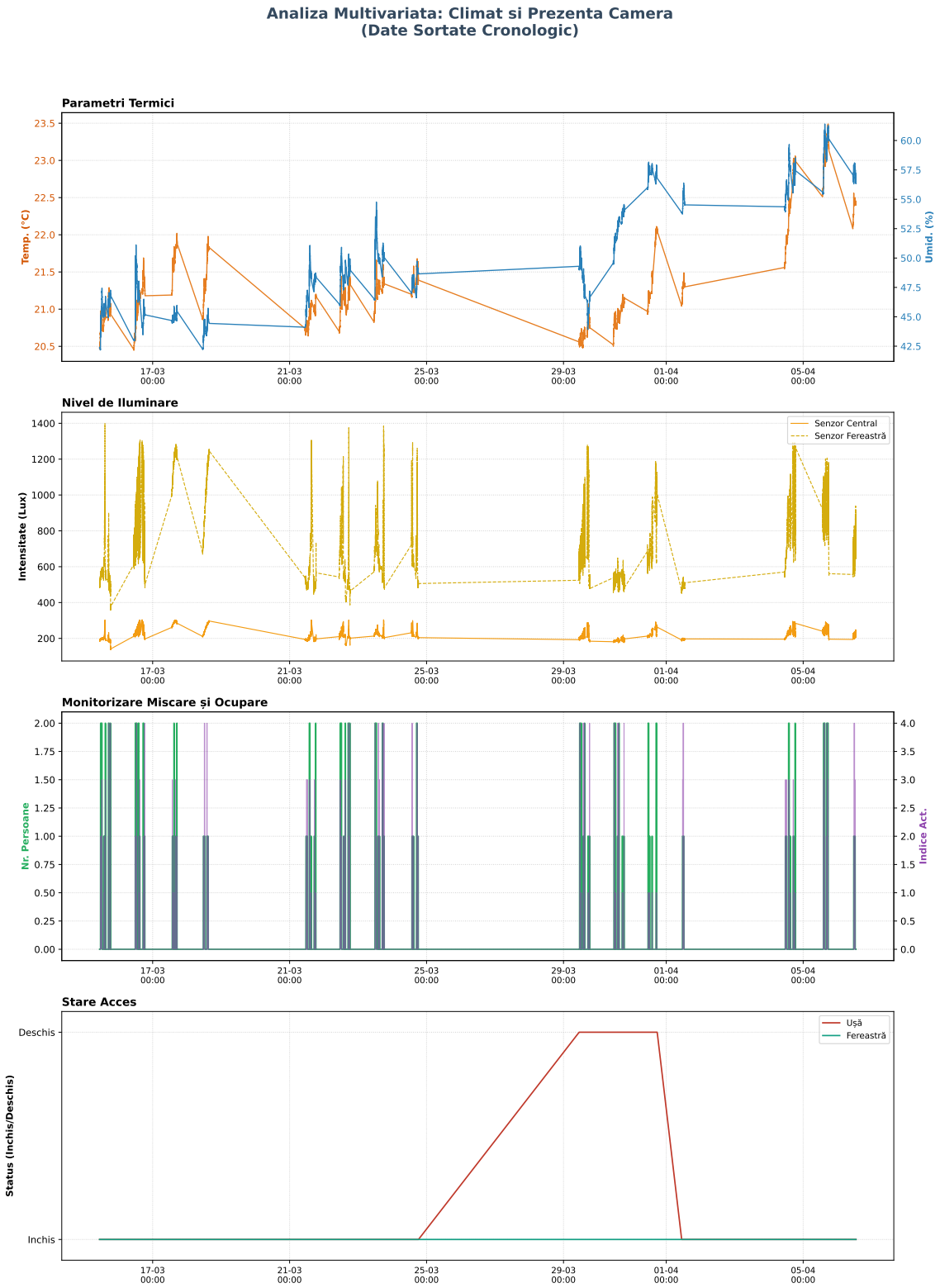

Figura 5.2: Vizualizarea seriei multivariate Room Climate. Graficul prezintă evoluția celor 8 variabile măsurate de senzorii de mediu interior.

5.12.3 Seria Twitter Volume — Compresia Standard

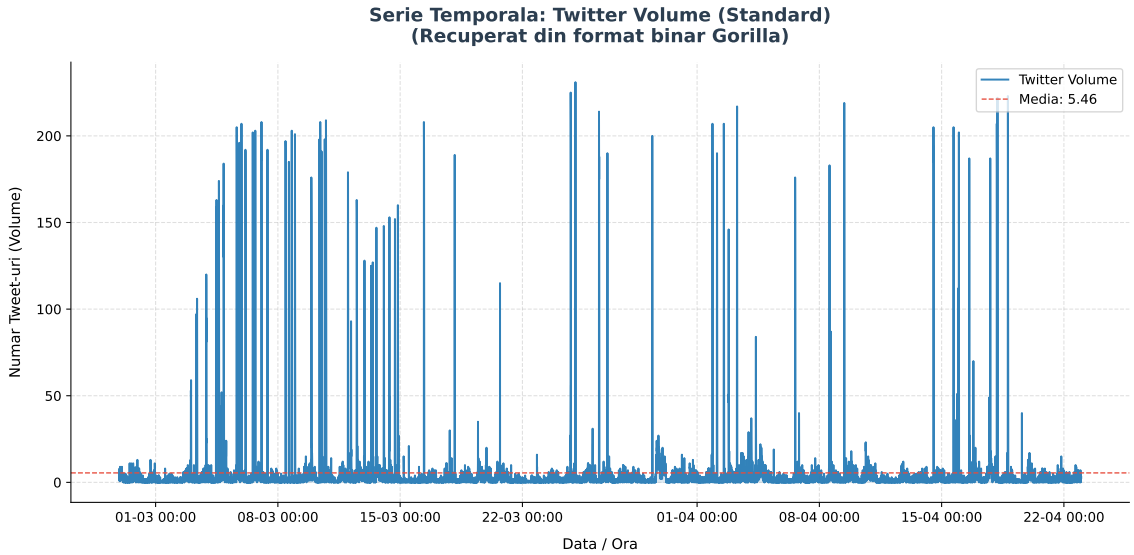

Figura 5.3: Vizualizarea seriei Twitter Volume procesată cu algoritmul Gorilla Standard. Graficul arată volumul de tweet-uri pe parcursul perioadei de colectare.

5.12.4 Seria Twitter Volume — Compresia Optimizată

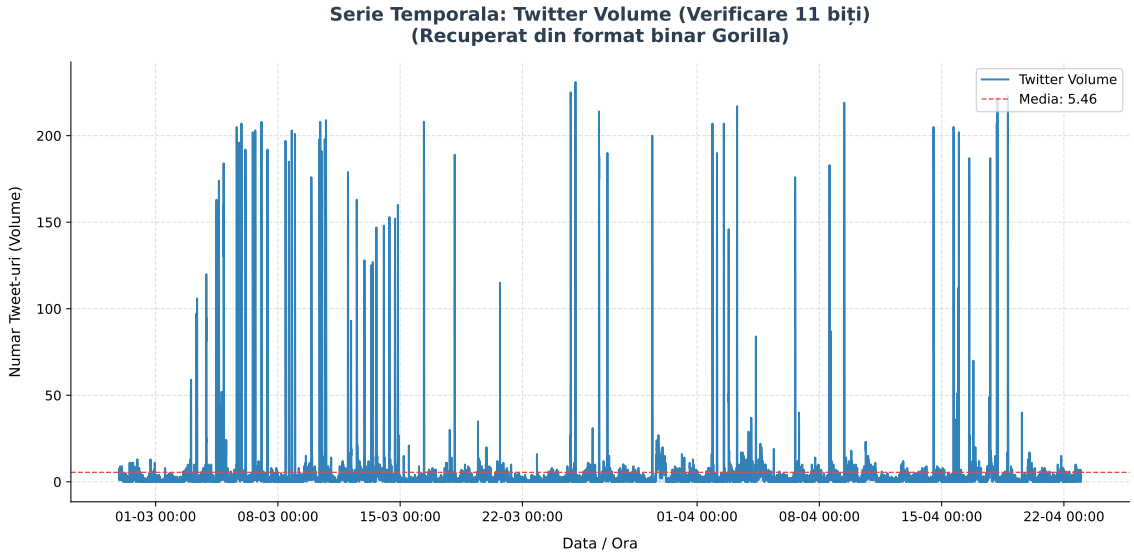

Figura 5.4: Vizualizarea seriei Twitter Volume procesată cu varianta Gorilla Optimizată. Datele decomprimate sunt identice cu cele din varianta standard, confirmând corectitudinea implementării.

5.12.5 Validarea integrității datelor — Twitter

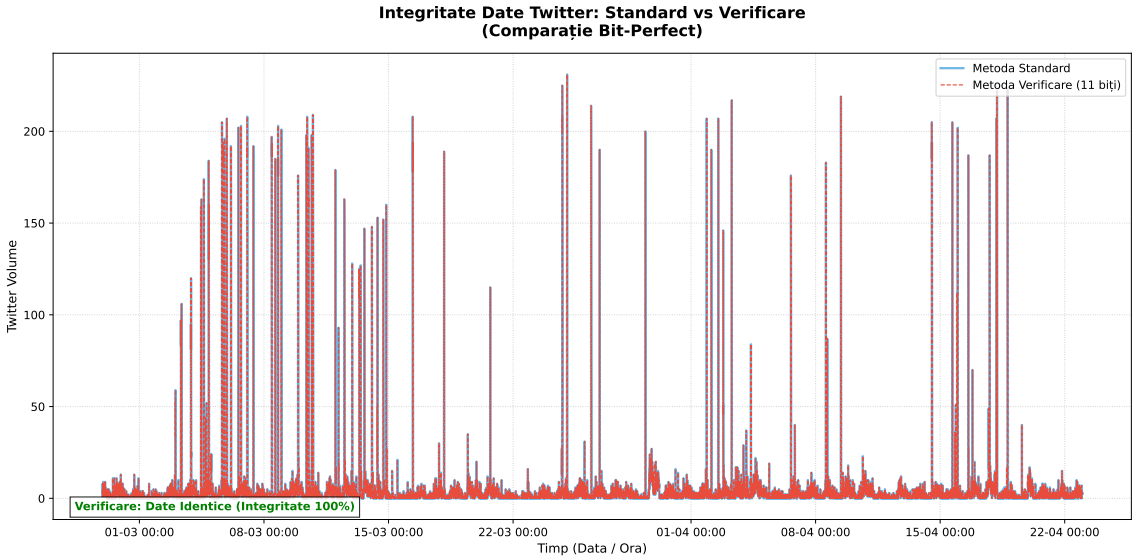

Figura 5.5: Comparația integrității datelor pentru seria Twitter Volume. Graficul suprapune datele originale cu cele decomprimate din ambele variante (Standard și Optimizată), demonstrând că compresia este *lossless* — toate valorile sunt recuperate exact.

Capitolul 6

Concluzii

6.1 Sinteza contribuțiilor

Prezentul proiect a implementat, analizat și **optimizat** algoritmul de compresie Gorilla pentru serii temporale. Principalele contribuții sunt:

6.1.1 Implementare completă în Python

A fost realizată o implementare funcțională a algoritmului Gorilla, incluzând:

- Module pentru operații la nivel de bit (`BitWriter`, `BitReader`);
- Compresie delta-of-delta pentru timestamp-uri cu codare variabilă;
- Compresie XOR pentru valori în virgulă mobilă cu refolosire de fereastră;
- Suport pentru serii univariate și multivariate;
- Organizare în blocuri de durată configurabilă (implicit 2 ore).

6.1.2 Optimizarea algoritmului: Condiția ferestrei de 11 biți

Contribuția originală a acestui proiect constă în identificarea și implementarea unei optimizări a condiției de refolosire a ferestrei XOR. Algoritmul original Gorilla refolosește fereastra anterioară ori de câte ori aceasta "încape" în fereastra curentă. Noi am demonstrat că această decizie nu este întotdeauna optimă.

Optimizarea propusă adaugă verificarea:

$$M_{anterior} - M_{curent} \leq 11 \tag{6.1}$$

Această condiție este **matematic optimă**: crearea unei ferestre noi costă exact 11 biți (5 pentru leading zeros + 6 pentru lungime), deci merită să o creăm doar dacă economia depășește acest overhead.

6.1.3 Documentare teoretică

Au fost prezentate în detaliu:

- Fundamentele matematice ale reprezentării în complement față de 2;
- Schema de codare variabilă pentru delta-of-delta (buckets de 1, 9, 12, 16, 36 biți);
- Mecanismul XOR și optimizarea prin refolosirea ferestrei de biți;
- Analiza matematică a trade-off-ului fereastră veche vs. fereastră nouă;
- Structura ierarhică a implementării (3 niveluri: biți, encodere, stocare).

6.1.4 Validare experimentală extinsă

Implementarea a fost testată pe **trei seturi de date** reale:

Dataset	Puncte	Gorilla Std	Gorilla Opt
CPU Load (univariat)	480	2.20x	2.37x
Twitter Volume (univariat)	15,866	6.59x	6.64x
Room Climate (8 variabile)	68,229	4.13x	4.20x

Tabela 6.1: Rate de compresie obținute

Toate testele au validat corectitudinea round-trip a compresiei (lossless, bit-perfect).

6.2 Rezultatele optimizării

Optimizarea propusă a demonstrat îmbunătățiri pe **toate** seturile de date testate:

Dataset	Economie (bytes)	Îmbunătățire
CPU Load	249 B	**7.14%**
Twitter Volume	262 B	0.68%
Room Climate	21,253 B	1.79%
Total	**21,764 B**	**1.77%**

Tabela 6.2: Îmbunătățirile aduse de varianta optimizată

Observație importantă: Nu există niciun caz în care varianta standard ar fi mai bună — optimizarea este **universal benefică**. Aceasta confirmă corectitudinea analizei matematice.

6.3 Concluzii tehnice

6.3.1 Eficiența delta-of-delta

Compresia timestamp-urilor este extrem de eficientă pentru date cu intervale regulate:

- Pentru Twitter (interval perfect de 5 minute): >99% comprimate la 1 bit;
- Pentru CPU și Room Climate: ∼85% comprimate la 1 bit;
- Media: 2–3 biți per timestamp, față de 64 biți necomprimat.

6.3.2 Eficiența XOR encoding

Compresia valorilor depinde de caracteristicile datelor:

- Valori constante sau aproape constante: 1 bit (XOR = 0);
- Valori cu variație lentă: beneficiază de refolosirea ferestrei (2 + M biți);
- Valori cu variație bruscă: necesită fereastră nouă (13 + M biți).

Optimizarea noastră reduce ineficiența în cazurile în care fereastra anterioară era semnificativ supradimensionată.

6.3.3 Avantajul stocării multivariate

Pentru serii cu multiple variabile per timestamp:

- Un singur stream de timestamp-uri pentru toate variabilele;
- Fiecare variabilă menține context XOR propriu;
- Economie de ∼3.7 MB pentru Room Climate (72 → 17 bytes/punct).

6.4 Limitări identificate

6.4.1 Date cu variabilitate mare

Algoritmul nu este optim pentru:

- Timestamp-uri aleatorii sau foarte neregulate;
- Valori care variază drastic între măsurători consecutive;
- Date criptate sau comprimate anterior (aspect aleatoriu).

6.4.2 Overhead pentru serii scurte

Pentru serii cu foarte puține puncte (<100), overhead-ul primelor valori (stocate integral pe 64 biți) domină, reducând eficiența compresiei.

6.4.3 Decodare secvențială

Decomprimarea necesită parcurgerea secvențială de la începutul blocului. Nu există acces aleatoriu la un punct specific fără a decoda toate punctele anterioare din bloc.

6.4.4 Implementare interpretată

Fiind scrisă în Python (limbaj interpretat), implementarea are performanță inferioară unei implementări native. Timpii de compresie observați ($\sim$2.7 secunde pentru 68,229 puncte × 8 variabile) ar putea fi reduși semnificativ într-un limbaj compilat.

6.5 Direcții de dezvoltare ulterioară

6.5.1 Optimizări de performanță

- **Implementare în C/Rust**: Port-area la un limbaj compilat pentru performanță de producție;
- **Paralelizare**: Blocurile independente pot fi comprimate/decomprimate în paralel.

6.5.2 Funcționalități adiționale

- **Partial decode**: Decodare doar a unui interval din bloc;
- **Indexare**: Structuri pentru acces rapid la anumite timestamp-uri;
- **Streaming API**: Interfață pentru procesare în flux a datelor;
- **Suport pentru missing values**: Gestionarea explicită a punctelor lipsă.

6.5.3 Extinderea optimizării

- **Analiză pe mai multe seturi de date**: Validarea optimizării pe date din alte domenii;
- **Threshold adaptiv**: Investigarea dacă pragul de 11 biți poate fi ajustat dinamic;

6.6 Concluzii

Algoritmul Gorilla reprezintă o soluție elegantă și eficientă pentru compresia seriilor temporale. Prin exploatarea inteligentă a proprietăților specifice ale acestui tip de date — periodicitatea timestamp-urilor și corelația temporală a valorilor — obține rate de compresie semnificative (2–7x în funcție de date) menținând caracterul lossless.

Contribuția principală a acestui proiect este optimizarea condiției de refolosire a ferestrei XOR, care aduce îmbunătățiri consistente (0.68%–7.14%) fără niciun dezavantaj. Această optimizare este **matematic optimă** și a fost **validată experimental** pe trei seturi de date diferite.

Implementarea realizată demonstrează fezabilitatea aplicării algoritmului (și a optimizării) pe date reale. Deși Python nu este limbajul optim pentru performanță maximă, claritatea codului a facilitat:

- Înțelegerea profundă a algoritmului;
- Identificarea oportunității de optimizare;

- Validarea corectitudinii prin teste round-trip.

Compresia seriilor de timp rămâne un domeniu activ de cercetare, cu aplicații în creștere odată cu proliferarea IoT și a sistemelor de monitorizare distribuite. Optimizarea prezentată în această lucrare demonstrează că algoritmii consacrați pot fi încă îmbunătățiți prin analiză atentă a cazurilor marginale.

Recomandare finală: Varianta optimizată a algoritmului Gorilla ar trebui adoptată în toate implementările, fiind superioară din punct de vedere matematic și confirmat experimental, fără niciun compromis.