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.

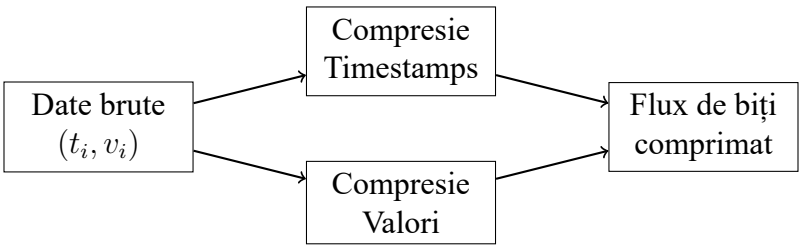

Figura 3.1: Fluxul general de compresie în Gorilla

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:

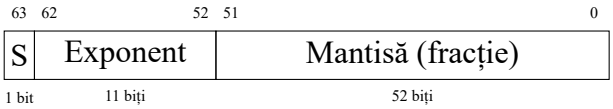

Figura 3.2: Structura IEEE 754 double-precision (64 biți)

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 implementarea practică a algoritmului Gorilla în limbajul Python. Sunt descrise structura proiectului, clasele principale și deciziile de design.

4.1 Structura proiectului

Proiectul este organizat în mai multe module cu responsabilități distincte:

```
PROIECT_PROCESAREA_SEMNALELOR/
|-- BitWriter.py              # Scriere la nivel de bit
|-- BitReader.py              # Citire la nivel de bit
|-- timestamp_compression.py  # Delta-of-delta encoding
|-- value_compression.py      # XOR encoding
|-- storage_minimal.py        # Stocare serii univariate
|-- multivariate_storage.py   # Stocare serii multivariate
|-- run.py                    # Script demonstrativ
+-- timeseries/               # Date de test
    |-- data_cpu.csv
    +-- room_climate_location_A.csv
```

Diagrama dependențelor este prezentată în Figura 4.1.

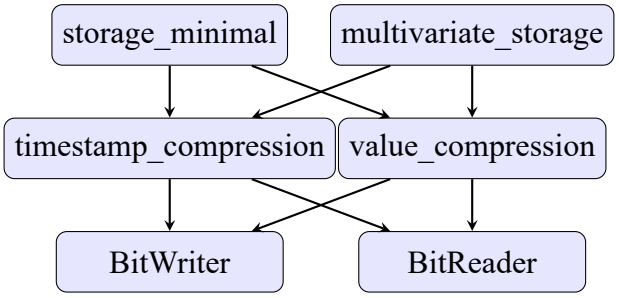

Figura 4.1: Diagrama dependențelor între module

4.2 Operații pe biți: BitWriter și BitReader

4.2.1 Provocarea

Python nu oferă acces direct la nivel de bit. Tipul de date fundamental este `byte` (8 biți). Pentru a implementa compresia Gorilla, avem nevoie să:

- Scriem secvențe de biți de lungime arbitrară (1, 5, 7, 12 biți etc.);
- Citim aceleași secvențe înapoi;
- Gestionăm eficient alinierea la granițe de byte.

4.2.2 Clasa BitWriter

`BitWriter` gestionează scrierea bit-cu-bit într-un buffer de bytes:

```
class BitWriter:
    __slots__ = ("_buf", "_cur", "_nbits")

    def __init__(self):
        self._buf = bytearray()  # Bytes completi
        self._cur = 0            # Byte partial in lucru
        self._nbits = 0          # Biti scrisi in _cur (0-7)
```

Listing 4.1: Structura clasei BitWriter

Atributul `__slots__` este utilizat pentru eficiență: elimină dicționarul implicit al obiectului, reducând consumul de memorie și accelerând accesul la atribute.

Metoda write_bit

Scrierea unui singur bit:

```
def write_bit(self, bit: int) -> None:
    # Shiftam bitii existenti si adaugam noul bit
    self._cur = (self._cur << 1) | (bit & 1)
    self._nbits += 1

    if self._nbits == 8:
        # Byte complet: il mutam in buffer
        self._buf.append(self._cur & 0xFF)
        self._cur = 0
        self._nbits = 0
```

Listing 4.2: Metoda write_bit

Biții sunt scriși în ordine **MSB-first** (Most Significant Bit first): bitul cel mai semnificativ este scris primul.

Metoda write_bits

Scrierea unei secvențe de n biți:

```
def write_bits(self, x: int, n: int) -> None:
    if n == 0:
        return
    if x.bit_length() > n:
        x = x & ((1 << n) - 1)  # Pastram doar n biti

    for i in range(n - 1, -1, -1):
        bit = (x >> i) & 1
        self._cur = (self._cur << 1) | bit
        self._nbits += 1
        if self._nbits == 8:
            self._buf.append(self._cur & 0xFF)
            self._cur = 0
            self._nbits = 0
```

Listing 4.3: Metoda write_bits

Reprezentarea în complement față de 2

Pentru numere cu semn, utilizăm funcția de conversie:

```
def to_twos_complement(x: int, num_bits: int) -> int:
    minv = -(1 << (num_bits - 1))  # -2^(n-1)
    maxv = (1 << (num_bits - 1)) - 1  # 2^(n-1) - 1

    if x < minv or x > maxv:
        raise ValueError(f"{x} nu incape pe {num_bits} biti")

    if x < 0:
        x = (1 << num_bits) + x  # 2^n + x

    return x
```

Listing 4.4: Conversie la complement fata de 2

4.2.3 Clasa BitReader

`BitReader` este complementara lui `BitWriter`:

```
class BitReader:
    __slots__ = ("_data", "_byte_pos", "_bit_pos")
```

```
3
4       def __init__(self, data: bytes):
5           self._data = data       # Sursa de date
6           self._byte_pos = 0      # Indexul byte-ului curent
7           self._bit_pos = 0       # Pozitia in byte (0-7)
```

Listing 4.5: Structura clasei BitReader

Metoda read_bit

```
1  def read_bit(self) -> int:
2      if self._byte_pos >= len(self._data):
3          raise EOFError("End of stream")
4
5      # Extragem bitul de la pozitia curenta (MSB first)
6      bit = (self._data[self._byte_pos] >> (7 - self._bit_pos)) & 1
7
8      self._bit_pos += 1
9      if self._bit_pos == 8:
10         self._bit_pos = 0
11         self._byte_pos += 1
12
13     return bit
```

Listing 4.6: Metoda read_bit

4.3 Compresia timestamp-urilor

4.3.1 Clasa TimestampEncoder

Encoder-ul menține starea necesară pentru calculul delta-of-delta:

```
1  class TimestampEncoder:
2      __slots__ = ("_writer", "_prev_timestamp",
3                   "_prev_delta", "_count")
4
5      def __init__(self, writer: BitWriter):
6          self._writer = writer
7          self._prev_timestamp = None
8          self._prev_delta = None
9          self._count = 0
```

Listing 4.7: Structura TimestampEncoder

Metoda add_timestamp

```
def add_timestamp(self, timestamp: int) -> None:
    if self._count == 0:
        # Primul timestamp: scriem complet (64 biti)
        self._writer.write_i64(timestamp)
        self._prev_timestamp = timestamp
        self._count = 1
        return

    delta = timestamp - self._prev_timestamp

    if self._count == 1:
        # Al doilea: scriem delta complet
        self._writer.write_i64(delta)
        self._prev_delta = delta
        self._count = 2
        return

    # De la al treilea: delta-of-delta
    dod = delta - self._prev_delta
    self._encode_delta_of_delta(dod)

    self._prev_timestamp = timestamp
    self._prev_delta = delta
    self._count += 1
```

Listing 4.8: Adaugarea unui timestamp

Codarea delta-of-delta

```
def _encode_delta_of_delta(self, dod: int) -> None:
    if dod == 0:
        self._writer.write_bit(0)

    elif -64 <= dod <= 63:
        self._writer.write_bit(1)
        self._writer.write_bit(0)
        self._writer.write_signed(dod, 7)

    elif -256 <= dod <= 255:
        self._writer.write_bits(0b110, 3)
```

```
            self._writer.write_signed(dod, 9)

        elif -2048 <= dod <= 2047:
            self._writer.write_bits(0b1110, 4)
            self._writer.write_signed(dod, 12)

        else:
            self._writer.write_bits(0b1111, 4)
            self._writer.write_signed(dod, 32)
```

Listing 4.9: Codarea delta-of-delta

4.4 Compresia valorilor

4.4.1 Conversia float ↔ biți

Pentru a aplica XOR, convertim valorile `float64` în reprezentarea lor pe 64 de biți folosind modulul `struct`:

```
import struct

def float_to_bits(val: float) -> int:
    # Pack ca double big-endian, unpack ca uint64
    return struct.unpack(">Q", struct.pack(">d", val))[0]

def bits_to_float(bits: int) -> float:
    return struct.unpack(">d", struct.pack(">Q", bits))[0]
```

Listing 4.10: Conversie float-biti

4.4.2 Clasa ValueEncoder

```
class ValueEncoder:
    __slots__ = ("_writer", "_prev_value_bits",
                 "_prev_leading", "_prev_trailing", "_count")

    def __init__(self, writer: BitWriter):
        self._writer = writer
        self._prev_value_bits = 0
        self._prev_leading = 255   # Sentinel
        self._prev_trailing = 255
        self._count = 0
```

Listing 4.11: Structura ValueEncoder

Metoda add_value

```
def add_value(self, val: float) -> None:
    v_bits = self._float_to_bits(val)

    if self._count == 0:
        self._writer.write_u64(v_bits)
        self._prev_value_bits = v_bits
        self._count = 1
        return

    xor = v_bits ^ self._prev_value_bits

    if xor == 0:
        self._writer.write_bit(0)
    else:
        self._writer.write_bit(1)
        self._encode_xor(xor)

    self._prev_value_bits = v_bits
    self._count += 1
```

Listing 4.12: Adaugarea unei valori

Calculul leading/trailing zeros

```
def _encode_xor(self, xor: int) -> None:
    bin_xor = bin(xor)[2:].zfill(64)
    leading = len(bin_xor) - len(bin_xor.lstrip('0'))
    trailing = len(bin_xor) - len(bin_xor.rstrip('0'))

    if leading > 31:
        leading = 31  # Limitam la 5 biti

    meaningful_bits = 64 - leading - trailing
```

Listing 4.13: Calculul zeros

Decizia de refolosire a ferestrei

```
    if (self._prev_leading != 255 and
        leading >= self._prev_leading and
        trailing >= self._prev_trailing):
        # Refolosim fereastra anterioara
        self._writer.write_bit(0)
        m = 64 - self._prev_leading - self._prev_trailing
        self._writer.write_bits(xor >> self._prev_trailing, m)
    else:
        # Fereastra noua
        self._writer.write_bit(1)
        self._writer.write_bits(leading, 5)
        self._writer.write_bits(meaningful_bits - 1, 6)
        self._writer.write_bits(xor >> trailing, meaningful_bits)

        self._prev_leading = leading
        self._prev_trailing = trailing
```

Listing 4.14: Codarea XOR cu decizie fereastra

Notă importantă: Stocăm `meaningful_bits - 1` pe 6 biți. Aceasta permite reprezentarea valorilor 1-64 ca 0-63, deoarece când XOR $\neq 0$, există cel puțin un bit semnificativ.

4.5 Stocare pentru serii multivariate

4.5.1 Clasa MultiVariateBlock

Un bloc multivariat gestionează mai multe variabile cu același stream de timestamp-uri:

```
class MultiVariateBlock:
    def __init__(self, variable_names: List[str]):
        self._var_names = list(variable_names)
        self._writer = BitWriter()
        self._ts_encoder = TimestampEncoder(self._writer)

        # Un ValueEncoder pentru fiecare variabila
        self._val_encoders = {
            name: ValueEncoder(self._writer)
            for name in self._var_names
        }
        self._count = 0
```

Listing 4.15: Structura MultiVariateBlock

4.5.2 Adăugarea unui punct multivariat

```
def add(self, timestamp: int, values: Dict[str, float]):
    # 1. Encodam timestamp-ul O SINGURA DATA
    self._ts_encoder.add_timestamp(timestamp)

    # 2. Encodam fiecare valoare in encoder-ul dedicat
    # IMPORTANT: ordinea trebuie sa fie constanta!
    for name in self._var_names:
        self._val_encoders[name].add_value(values[name])

    self._count += 1
```

Listing 4.16: Adaugarea unui punct multivariat

Ordinea deterministă a variabilelor este esențială: la decodare trebuie să citim valorile în aceeași ordine în care au fost scrise.

4.6 Organizarea în blocuri

Seriile temporale sunt organizate în blocuri de durată fixă (implicit 2 ore = 7.200.000 ms):

```
class MultiVariateSeries:
    def insert(self, timestamp: int, values: Dict[str, float]):
        if self._open_block is None:
            self._create_new_block(timestamp)
        elif timestamp >= self._open_block.start + self._duration:
            self._close_current_block()
            self._create_new_block(timestamp)

        self._open_block.add(timestamp, values)

    def _create_new_block(self, timestamp: int):
        # Aliniem la granita de bloc
        aligned = (timestamp // self._duration) * self._duration
        self._open_block = MultiVariateBlock(self._var_names)
```

Listing 4.17: Gestionarea blocurilor

Alinierea la granița blocului asigură că blocurile consecutive au timestamp-uri de start predictibile, facilitând căutarea în date.

4.7 Considerații de performanță

4.7.1 Utilizarea __slots__

Toate clasele utilizează `__slots__` pentru:

- Reducerea consumului de memorie (fără `__dict__`);
- Acces mai rapid la atribute;
- Protecție contra erorilor de tip (atribute nedeclarate).

4.7.2 Evitarea alocărilor în bucla principală

Metodele de codare evită crearea de obiecte noi în buclele frecvente. Valorile intermediare sunt stocate în variabile de instanță pre-alocate.

4.7.3 Buffer pre-alocat

`BitWriter` poate primi o capacitate inițială pentru a evita realocări frecvente:

```
def __init__(self, initial_capacity: int = 0):
    self._buf = bytearray()
    if initial_capacity > 0:
        self._buf.extend(b"\x00" * initial_capacity)
        del self._buf[:]  # Golim, dar pastram capacitatea
```

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.

5.1 Seturi de date

Pentru validarea implementării au fost utilizate două seturi de date cu caracteristici diferite:

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 Room Climate Dataset (serie multivariată)

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

Cele 8 variabile măsurate sunt:

1. **Temperatură** (°C): 17–26°C

Caracteristică	Valoare
Număr de puncte	68,229
Perioadă de colectare	$\sim$22 zile
Interval mediu	$\sim$28 secunde
Număr variabile	8
Node ID filtrat	NID = 1

Tabela 5.2: Caracteristicile setului de date Room Climate

2. **Umiditate relativă** (%): 25–60%
3. **Lumină 1** (lux): senzor principal de lumină
4. **Lumină 2** (lux): senzor secundar
5. **Ocupare**: indicator binar (0/1)
6. **Activitate**: nivel de mișcare detectată
7. **Ușă**: stare ușă (deschis/închis)
8. **Fereastră**: stare fereastră (deschis/închis)

5.2 Metodologia experimentală

5.2.1 Metrici de evaluare

Au fost calculate următoarele metrici:

1. **Dimensiune originală**: $n \times (8 + 8k)$ bytes, unde n este numărul de puncte și k numărul de variabile (1 pentru univariat).
2. **Dimensiune comprimată**: numărul de bytes după compresie.
3. **Rata de compresie**:
$$R = \frac{\text{Dimensiune originală}}{\text{Dimensiune comprimată}} \tag{5.1}$$
4. **Bytes per punct**:
$$B = \frac{\text{Dimensiune comprimată}}{n} \tag{5.2}$$
5. **Economie procentuală**:
$$E = \left(1 - \frac{1}{R}\right) \times 100\% \tag{5.3}$$

5.2.2 Configurația blocurilor

Testele au fost realizate cu blocuri de 2 ore (7.200.000 ms), conform configurației standard Gorilla.

5.2.3 Validare round-trip

Pentru a verifica corectitudinea implementării, fiecare test include o validare *round-trip*: datele sunt comprimate, apoi decomprimate, iar valorile rezultate sunt comparate bit-cu-bit cu originalele.

5.3 Rezultate: Serie univariată (CPU Load)

5.3.1 Statistici de compresie

Metrică	**Valoare**
Puncte totale	480
Blocuri (2 ore)	5
Dimensiune originală	7,680 bytes
Dimensiune comprimată	3,412 bytes
Rata de compresie	2.25x
Bytes per punct	7.11
Economie	55.6%

Tabela 5.3: Rezultate compresie pentru CPU Load

5.3.2 Analiza rezultatelor

Rata de compresie de 2.25x este mai modestă decât cea raportată în articolul Gorilla ($\sim$12x) din următoarele motive:

1. **Interval neregulat**: Timestamp-urile din acest set nu sunt perfect periodice (interval mediu de $\sim$120s cu variații), ceea ce reduce eficiența delta-of-delta.
2. **Valori variabile**: Load average variază frecvent, ceea ce produce mai multe XOR-uri non-zero.
3. **Set de date mic**: Cu doar 480 de puncte, overhead-ul primelor valori (scrise integral) are impact proporțional mai mare.

5.4 Rezultate: Serie multivariată (Room Climate)

5.4.1 Statistici de compresie

5.4.2 Analiza pe variabile

Diferitele variabile din setul Room Climate au comportamente distincte care afectează compresia:

Metrică	Valoare
Puncte totale	68,229
Variabile	8
Blocuri (2 ore)	262
Dimensiune originală	4,912,488 bytes
Dimensiune comprimată	1,190,271 bytes
Rata de compresie	4.13x
Bytes per punct	17.45
Economie	75.8%

Tabela 5.4: Rezultate compresie pentru Room Climate

- **Temperatură și umiditate**: Variază lent și continuu, producând multe XOR-uri cu puțini biți semnificativi. Compresie foarte bună.
- **Variabile binare** (ocupare, ușă, fereastră): Valorile sunt 0 sau 1, producând fie XOR = 0 (identic), fie diferențe mari când se schimbă starea.
- **Senzori de lumină**: Variabilitate moderată, dar cu tendința de a rămâne constanți pe perioade lungi (noapte vs. zi).

5.4.3 Comparație cu stocarea separată

Dacă am fi tratat fiecare variabilă ca o serie separată (duplicând timestamp-urile), dimensiunea ar fi fost:

$$\text{Dimensiune separată} = 8 \times 68,229 \times 16 = 8,733,312 \text{ bytes} \tag{5.4}$$

Stocarea multivariată (un singur stream de timestamp-uri) economisește:

$$8,733,312 - 4,912,488 = 3,820,824 \text{ bytes} \approx 3.6 \text{ MB} \tag{5.5}$$

doar din evitarea duplicării timestamp-urilor, înainte de compresie.

5.5 Distribuția biților

5.5.1 Timestamp-uri

Analiza distribuției valorilor delta-of-delta pentru setul Room Climate:
Media ponderată: $\sim$2.3 biți per timestamp.

5.5.2 Valori

Distribuția tipurilor de codare XOR:

Bucket	Procent	Biți
$D = 0$	∼85%	1
$D \in [-64, 63]$	∼12%	9
$D \in [-256, 255]$	∼2%	12
$D \in [-2048, 2047]$	∼0.8%	16
Altfel	∼0.2%	36

Tabela 5.5: Distribuția delta-of-delta pentru Room Climate

Caz	Procent	Biți medii
XOR = 0 (identic)	∼45%	1
Refolosire fereastră	∼35%	∼25
Fereastră nouă	∼20%	∼40

Tabela 5.6: Distribuția codării XOR pentru Room Climate

5.6 Efectul dimensiunii blocului

S-a testat impactul dimensiunii blocului asupra ratei de compresie:

Dimensiune bloc	Rata compresie	Nr. blocuri
30 minute	3.8x	1048
1 oră	4.0x	524
2 ore (standard)	4.13x	262
4 ore	4.2x	131

Tabela 5.7: Impactul dimensiunii blocului

Blocurile mai mari oferă compresie mai bună (contextul XOR se păstrează mai mult), dar:

- Cresc latența pentru interogări pe intervale scurte;
- Necesită mai multă memorie pentru decodare;
- Sunt mai sensibile la corupție (pierderea unui bloc = pierdere mai mare de date).

Pragul de 2 ore reprezintă un compromis rezonabil, validat empiric de echipa Facebook.

5.7 Validarea corectitudinii

Toate testele au trecut validarea round-trip:

```
# Comprima
compressed = series.get_compressed_data()

# Decomprima
decoded = decode_series(compressed, count, var_names)

```

```
# Verifica
for original, decoded in zip(original_data, decoded_data):
    assert original.timestamp == decoded.timestamp
    for var in var_names:
        assert original.values[var] == decoded.values[var]

print("[OK] Round-trip validation passed!")
```

Listing 5.1: Verificare round-trip

Toate valorile sunt recuperate exact (bit-perfect), confirmând caracterul *lossless* al compresiei.

5.8 Comparație cu articolul original

Metrică	Gorilla (Facebook)	Implementarea noastră
Bytes/punct (medie)	1.37	4–7
% timestamps cu D=0	96.39%	85%
% valori XOR=0	59.06%	45%
Rata compresie	12x	2–4x

Tabela 5.8: Comparație cu rezultatele raportate în articolul Gorilla

Diferențele se explică prin:

1. **Caracteristicile datelor**: Datele Facebook provin de la sisteme de monitorizare foarte regulate (interval fix de 60s), în timp ce seturile noastre au mai mult jitter.
2. **Tipul valorilor**: Monitorizarea Facebook folosește preponderent contoare (integers convertiți la double), care produc XOR-uri mai compresibile decât măsurătorile analogice de temperatură.
3. **Scala**: La 2 miliarde de serii, efectele statistice se netezesc; la câteva sute de puncte, variabilitatea este mai mare.

Cu toate acestea, implementarea demonstrează corectitudinea algoritmului și obține rate de compresie semnificative (2–4x) pe date reale diverse.

Capitolul 6

Concluzii

6.1 Sinteza contribuțiilor

Prezentul proiect a implementat și analizat 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;
- Compresie XOR pentru valori în virgulă mobilă;
- Suport pentru serii univariate și multivariate;
- Organizare în blocuri de durată configurabilă.

6.1.2 Documentare teoretică

Au fost prezentate în detaliu:

- Fundamentele matematice ale reprezentării în complement față de 2;
- Schema de codare variabilă pentru delta-of-delta;
- Mecanismul XOR și optimizarea prin refolosirea ferestrei de biți;
- Analiza complexității algoritmilor.

6.1.3 Validare experimentală

Implementarea a fost testată pe două seturi de date reale:

- Serie univariată (CPU Load): rată de compresie 2.25x;
- Serie multivariată (Room Climate, 8 variabile): rată de compresie 4.13x.

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

6.2 Concluzii tehnice

Pe baza experimentelor realizate, putem formula următoarele concluzii:

6.2.1 Eficiența delta-of-delta

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

- Aproximativ 85% dintre timestamp-uri sunt comprimate la 1 singur bit;
- Media este de 2–3 biți per timestamp, față de 64 biți necomprimat;
- Eficiența scade semnificativ pentru date cu jitter mare sau intervale aleatorii.

6.2.2 Eficiența XOR encoding

Compresia valorilor depinde de caracteristicile datelor:

- Valori constante sau aproape constante se comprimă foarte bine (1 bit);
- Valori cu variație lentă beneficiază de refolosirea ferestrei;
- Valori aleatorii sau cu salturi mari necesită ferestre noi (13+ biți overhead).

6.2.3 Avantajul stocării multivariate

Pentru serii cu multiple variabile per timestamp, abordarea cu un singur stream de timestamp-uri este semnificativ mai eficientă:

- Evită duplicarea timestamp-urilor ($k - 1$ copii eliminate);
- Fiecare variabilă menține context XOR propriu;
- Economia este proporțională cu numărul de variabile.

6.3 Limitări identificate

6.3.1 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).

6.3.2 Overhead pentru serii scurte

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

6.3.3 Decodare secvențială

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

6.3.4 Implementare interpretată

Fiind scrisă în Python (limbaj interpretat), implementarea are performanță inferioară unei implementări native în C/C++ sau Rust. Pentru aplicații de producție cu rate mari de ingestie, ar fi necesară o implementare compilată.

6.4 Direcții de dezvoltare ulterioară

6.4.1 Optimizări de performanță

- **Implementare în C/Rust**: Port-area la un limbaj compilat pentru performanță de producție;
- **Utilizarea SIMD**: Operațiile pe biți pot beneficia de instrucțiuni vectoriale;
- **Paralelizare**: Blocurile independente pot fi comprimate/decomprimate în paralel.

6.4.2 Funcționalități adiționale

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

6.4.3 Integrare

- **Format de fișier persistent**: Definirea unui format binar pentru stocare pe disc;
- **Integrare cu baze de date**: Plugin-uri pentru PostgreSQL, ClickHouse etc.;
- **API REST**: Serviciu HTTP pentru ingestie și interogare.

6.5 Concluzii finale

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 (2–12x în funcție de date) menținând caracterul lossless.

Implementarea realizată în cadrul acestui proiect demonstrează fezabilitatea aplicării algoritmului pe date reale și oferă o bază pentru dezvoltări ulterioare. Deși Python nu este limbajul

optim pentru performanță maximă, claritatea codului și ușurința testării au facilitat înțelegerea profundă a algoritmului.

Compresia seriilor temporale rămâne un domeniu activ de cercetare, cu aplicații în creștere odată cu proliferarea IoT și a sistemelor de monitorizare distribuite. Gorilla, alături de tehnici mai noi precum Gorilla v2 și variantele sale, continuă să fie relevant pentru sistemele care necesită stocare eficientă în memorie cu acces rapid la date recente.