indexed_gzip: tajniki DEFLATE oraz losowy dostęp do skompresowanych danych

Szacuje się, że ludzkość generuje kwintyliony bajtów (10^{30}) danych dziennie [3]. Gdyby nie techniki kompresji danych, liczby te byłyby co najmniej kilkukrotnie większe. Kompresja danych jest obecnie tak powszechna, że trudno znaleźć miejsce, gdzie nie jest wykorzystywana. Kompresja wiąże się z pewnymi kosztami, jednak korzyści z niej płynące często przewyższają niedogodności.

W poniższym artykule przybliżymy algorytm stojący za formatem archiwów GZIP. Zaprezentujemy również sposób dekompresji fragmentów archiwów bez przetwarzania ich w całości. Dodatkowo przedstawimy przykład wykorzystania zdobytej wiedzy w praktycznym rozwiązaniu oszczędzającym czas procesora i przestrzeń dyskową.

KOMPRESJA DANYCH

Kompresja danych była stosowana długo przed powstaniem komputerów, jakie obecnie znamy. Już w 1838 roku w kodzie Morse'a został wykorzystany fakt, że w języku angielskim literki „e" i „t" występują częściej niż pozostałe, więc tym literkom przyporządkowane zostały krótsze symbole [11]. 110 lat później Claude Shannon wyznaczył teoretyczny limit bezstratnej kompresji danych, wiążąc go z ich entropią [12]. Do dzisiaj algorytmy kompresji wykorzystują fakt, iż dane generowane m.in. przez ludzi mają relatywnie niską entropię. Jednym z tych algorytmów jest opisany w artykule DEFLATE, który stoi za popularnym formatem archiwów GZIP. Aby zobrazować jego skuteczność, można porównać rezultaty kompresji dwóch rodzajów danych:

1. tekstu pisanego (autor zdecydował się na książkę autorstwa Janusza Zajdla pt. „Paradyzja"),
2. pliku z liczbami pseudo-losowymi.

W Listingu 1 przedstawiono takie porównanie. Tekst książki udało się skompresować około 2.6 raza, a kompresja danych pseudo-losowych okazała się nieskuteczna. W jaki sposób algorytmowi DEFLATE udało się uzyskać takie wyniki? Poza odpowiedzią na to pytanie w artykule zademonstrowana zostanie również technika losowego dostępu do dowolnej części skompresowanych danych bez dekompresji całego pliku.

W części związanej z losowym dostępem wykorzystana zostanie implementacja *zlib/zran.c* [0] biblioteki zlib opakowana w paczkę Pythona indexed_gzip [1]. Warto zwrócić uwagę, że implementacje indeksowanego GZIP dostępne są również dla innych języków – np. jzran dla Javy [2].

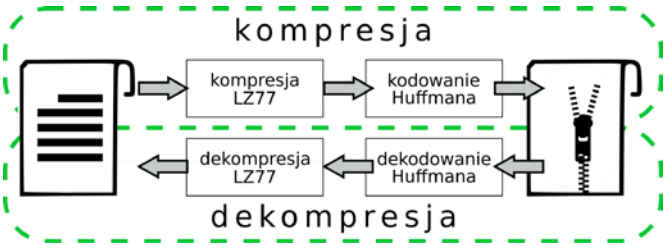
Listing 1. Porównanie stopnia kompresji danych o niskiej i wysokiej entropii

```
# Zliczanie znaków oryginalnego tekstu
$ wc -c paradyzja.txt
307306 paradyzja.txt

# Zliczanie znaków skompresowanego tekstu
$ gzip -c paradyzja.txt | wc -c
119361

# Zliczanie znaków skompresowanego pliku z losowo
# wygenerowanymi danymi.
$ dd if=/dev/urandom bs=1 count=307306 status=none \
  | gzip -c | wc -c
307374
```

CO POD MASKĄ KRYJE GZIP?

GZIP (od GNU zip) to nazwa programu oraz format pliku, na który składają się nagłówek oraz dane skompresowane algorytmem DEFLATE [4]. Właściwa jego implementacja zawiera się np. w bibliotece zlib. DEFLATE to dwuetapowy algorytm kompresji i dekompresji, którego motywem powstania były m.in problemy patentowe związane z wcześniej używanym programem *compress*.

DEFLATE składa się z dwóch głównych komponentów: kodowania Huffmana (ang. *Huffman coding*) oraz algorytmu LZ77 (Lempel-Ziv) (Rysunek 1).



Rysunek 1. Wysokopoziomowe kroki algorytmu DEFLATE

Rzeczywista implementacja GZIP jest skomplikowana, a wiele aspektów kompresji/dekompresji można parametryzować. Poniżej zamieszczony opis należy zatem traktować z przymrużeniem oka – nie służy on bowiem wyczerpującemu opisaniu wszystkich detali, lecz przybliżeniu czytelnikowi tematu.

LZ77

Tekst książki „Paradyzja" pozwala, jak większość tekstu pisanego, na uzyskanie dobrego współczynnika kompresji ze względu na powtarzające się w kółko słowa. Współczynnik kompresji to stosunek rozmiaru danych skompresowanych do wejściowych. W wymienionej książce znajduje się około 39 tysięcy słów. Unikalnych jest już jednak tylko około 11 tysięcy. Do najczęściej występujących słów należą takie jak „się", „w", „nie", „jest" czy „Rinah" (jeden z bohaterów).

Algorytm LZ77 oraz część jego pochodnych wykorzystują niską entropię takich danych i ich powtarzalność. Część danych, które się dotychczas pojawiły, zapamiętywana jest w słowniku i może zostać wykorzystana do kompresji reszty. Algorytm, przetwarzając fragment danych wejściowych, szuka innych fragmentów w słowniku, które współdzielą prefiks z danymi wejściowymi, i w przypadku znalezienia dopasowania emituje specjalną informację, którą można przetłumaczyć na „następne n kompresowanych bajtów jest identyczne z tym, co było przetwarzane k pozycji temu". Występowanie dłuższych prefiksów naturalnie przekłada się na lepszy współczynnik kompresji.

Niestety, ze względu na skończoną ilość RAM wczytywanie całego pliku do pamięci byłoby mało praktyczne lub niemożliwe. Z tego powodu algorytmy z rodziny LZ są strumieniowe. W tym celu wykorzystują tzw. pływające okienka (bufory). Podczas kompresji kolejne bajty pliku wejściowego przez te bufory przechodzą. W LZ77 występują dwa rodzaje buforów wejściowych:

» bufor słownikowy (ang. *search buffer*, czasami nazywany również *dictionary buffer*),
» bufor podglądu (ang. *lookahead buffer*).

Porcje danych trafiają najpierw do bufora podglądu (którego zawartość można rozumieć jako „co będzie niedługo kompresowane?"). Następnie algorytm szuka najdłuższego prefiksu bufora podglądu, który występuje w słowniku (szczególny przypadek problemu *longest common substring* [5]). Po każdym takim kroku emitowana jest tzw. krotka dopasowania (ang. *match*), która składa się z trzech elementów (k, n, c), gdzie:

» k – odległość względem końca bufora szukania, gdzie dopasowanie się zaczyna,
» n – długość dopasowania,
» c – kolejny bajt w buforze podglądu, który należy umieścić zaraz po dopasowaniu podczas dekompresji.

Istnieją oczywiście sytuacje, w których w słowniku nie znajduje się niepusty prefiks bufora podglądu – wówczas kompresor emituje krotkę dopasowania z wartościami k oraz n równymi 0.

Poniżej umieszczono przykład, który ułatwia zrozumienie algorytmu LZ77. Fraza, która została wybrana do przykładu, to:

Król Karol kupił królowej Karolinie korale koloru koralowego.

Dla uproszczenia ograniczono się do małych znaków, a te polskie zmieniono na ich odpowiedniki w ASCII:

krol␣karol␣kupil␣krolowej␣karolinie␣korale␣koloru␣koralowego.

W praktyce wielkości buforów są spore (np. 32 kB na każdy). Ze względu na ograniczone miejsce w przedstawionym przykładzie przyjmuje się dużo mniejsze bufory słownikowe i podglądu: odpowiednio 10 i 7 bajtów. Na Rysunku 2 przedstawione zostały 4 przykładowe kroki algorytmu. Tabela 0 zawiera wszystkie kroki z buforem słownikowym o rozmiarze 31.

Rysunek 2. Przykładowe kroki algorytmu LZ77. Żółtym kolorem oznaczono pasujący prefiks, a zielonym dokładany znak

bufor słownikowy																												bufor podglądu							krotka dopasowania	kodowany łańcuch
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	2	3	4	5	6	7	
																															k	r	o	l		k	a	(0, 0, k)	k
																														k	r	o	l		k	a	r	(0, 0, r)	r
																													k	r	o	l		k	a	r	o	(0, 0, o)	o
																												k	r	o	l		k	a	r	o	l	(0, 0, l)	l
																											k	r	o	l		k	a	r	o	l		(0, 0, ␣)	␣
																										k	r	o	l		k	a	r	o	l		k	(5, 1, a)	ka
																								k	r	o	l		k	a	r	o	l		k	u	p	(6, 5, u)	rol␣ku
																		k	r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	(0, 0, p)	p
																	k	r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	(0, 0, i)	i
																k	r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	(12, 3, r)	l␣kr
												k	r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		(17, 2, o)	olo
									k	r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	(0, 0, w)	w
								k	r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	(0, 0, e)	e
							k	r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	l	(0, 0, j)	j
						k	r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	l	i	(21, 6, i)	␣karoli
r	o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	l	i	n	i	e		k	o	r	(0, 0, n)	n
o	l		k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	l	i	n	i	e		k	o	r	a	(19, 1, e)	ie
	k	a	r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	l	i	n	i	e		k	o	r	a	l	e	(31, 2, o)	␣ko
r	o	l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	l	i	n	i	e		k	o	r	a	l	e		k	o	(31, 1, a)	ra
l		k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	l	i	n	i	e		k	o	r	a	l	e		k	o	l	o	(31, 1, e)	le
k	u	p	i	l		k	r	o	l	o	w	e	j		k	a	r	o	l	i	n	i	e		k	o	r	a	l	e		k	o	l	o	r	u	(7, 3, l)	␣kol
l		k	r	o	l	o	w	e	j		k	a	r	o	l	i	n	i	e		k	o	r	a	l	e		k	o	l	o	r	u		k	o	r	(9, 2, u)	oru
r	o	l	o	w	e	j		k	a	r	o	l	i	n	i	e		k	o	r	a	l	e		k	o	l	o	r	u		k	o	r	a	l	o	(14, 6, o)	␣koralo
	k	a	r	o	l	i	n	i	e		k	o	r	a	l	e		k	o	l	o	r	u		k	o	r	a	l	o	w	e	g	o	.			(0, 0, w)	w
k	a	r	o	l	i	n	i	e		k	o	r	a	l	e		k	o	l	o	r	u		k	o	r	a	l	o	w	e	g	o	.				(23, 1, g)	eg
r	o	l	i	n	i	e		k	o	r	a	l	e		k	o	l	o	r	u		k	o	r	a	l	o	w	e	g	o	.						(30, 1, .)	o.

Tabela 0. Wszystkie kroki algorytmu LZ77 dla przedstawionego przykładu

W każdym kroku w pierwszej kolejności dane trafiają do bufora podglądu, a później do bufora słownikowego. W pierwszym kroku z Rysunku 2 w buforze słownikowym na 6 pozycji znajduje się prefiks bufora podglądu o długości 5 znaków („rol␣k"). Po dołożeniu następnego znaku z bufora podglądu algorytm emituje krotkę dopasowania (6,5,'u'). W następnych dwóch krokach zawartość bufora podglądu rozpoczyna się od liter „p" oraz „i", które wcześniej nie wystąpiły, więc emitowane krotki dopasowań mają k i n ustawione na 0.

Jak widać, idea strumieniowej kompresji LZ77 nie jest bardzo skomplikowana. Oczywiście faktyczna implementacja niesie za sobą zdecydowanie więcej szczegółów, głównie ze względu na wydajność (więcej informacji znajduje się w ramce „Po trupach do wydajności").

Wynikiem działania algorytmu LZ77 są dopasowania, które następnie są kodowane algorytmem Huffmana. Algorytm ten opisany jest w dalszej części artykułu.

Kodowanie huffmana

Kodowanie Huffmana samo w sobie też jest sposobem na kompresję danych. Algorytm DEFLATE wykorzystuje je jako drugi krok kompresji. Idea kryjąca się za tym kodowaniem również nie jest skomplikowana, więc dla bardziej kompletnego wyobrażenia całego procesu także zostanie opisana.

Tym razem jako przykład wykorzystana zostanie fraza „abrakadabra". We frazie tej literka „a" występuje aż 5 razy, literki „r" i „b" występują po 2 razy, a literki „k" i „d" po jednym razie.

Kodowanie Huffmana polega na wyliczeniu częstości występowania danych symboli w tekście oraz takiego przyporządkowania ich do binarnych ciągów, aby najczęściej występującym symbolom przypisane były najkrótsze ciągi.

Długość hasła „abrakadabra" to 11. Prawdopodobieństwa wystąpienia literek przedstawiono w Tabeli 2.

literka	wystąpienia	prawdopodobieństwo (zaokrąglone)
a	5	45.(45)%
b	2	18.(18)%
r		
d	1	9.(09)%
k		

Tabela 2. Prawdopodobieństwa wystąpień literek we frazie „abrakadabra"

Na podstawie tych prawdopodobieństw algorytm buduje tzw. drzewo Huffmana, które będzie wykorzystane do kodowania danych wejściowych.

Drzewo Huffmana jest drzewem binarnym, którego liście reprezentują symbole pochodzące z danych wejściowych. Każdy węzeł z tego drzewa z kolei ma przypisane zsumowane prawdopodobieństwo wystąpienia któregokolwiek symbolu z poddrzewa. Węzły i liście w drzewie są tak ułożone, aby najczęściej występujące symbole w danych wejściowych znalazły się najbliżej korzenia. Dzięki temu długość ścieżki od korzenia do liścia danego symbolu jest odwrotnie proporcjonalna do prawdopodobieństwa wystąpienia tego symbolu.

Jeden z prostszych algorytmów, który konstruuje drzewo Huffmana, wykorzystuje kolejkę priorytetową. Początkowo dla każdej występującej litery tworzone jest drzewo z jednym liściem. Następnie zostają one umieszczone w kolejce priorytetowej, w której wysoki priorytet mają drzewa z najniższym prawdopodobieństwem.

W każdym kroku z kolejki pobierane są 2 pierwsze elementy (o ile jeszcze tyle jest), które są ze sobą scalane. Scalanie polega na połączeniu elementów korzeniem, którego prawdopodobieństwo jest sumą prawdopodobieństw łączonych poddrzew. Na przykład literki „d" i „k" z Tabeli 2 mają prawdopodobieństwo 9%. Po ich scaleniu korzeń je łączący będzie miał 18%. Po scaleniu takie drzewo wraca z powrotem do kolejki.

Po utworzeniu całego drzewa gałęziom lewym oraz prawym przypisywane są bity 0 albo 1. Kodowanie symbolu polega na zapisaniu ścieżki od korzenia do liścia za pomocą tych bitów. Dekodowanie polega na przechodzeniu po drzewie zgodnie z odczytywanymi bitami. Przykładowe drzewo Huffmana dla frazy „abrakadabra" przedstawiono na Rysunku 3.

Po trupach do wydajności

Kompresja i dekompresja są tak kluczowymi operacjami w świecie IT, że algorytmy (de)kompresji oraz ich implementacje stosują wiele sztuczek zwiększających wydajność.

Na przykład jednym z opisanych kroków LZ77 jest znalezienie najdłuższego wspólnego prefiksu. Wydawałoby się, że wystarczy wykorzystać algorytm rozwiązujący problem LCS [5], przekazując mu jako argumenty dwa bufory. W praktyce okazuje się m.in., że:

- dla uzyskania lepszych rezultatów dopuszczona jest możliwość rozpięcia prefiksu pomiędzy buforami (tj. prefiks zaczyna się w buforze słownikowym, a kończy w buforze podglądu),
- parametry kompresji mogą ograniczać maksymalny najdłuższy prefiks. Na przykład program GZIP umożliwia wiele tzw. poziomów kompresji, w tym takie, które uzyskują gorsze współczynniki kompresji na rzecz wydajności [6],
- stosowanie techniki leniwych dopasowań (ang. *lazy matching*) pozwala znajdować dłuższe dopasowania. Polega ona na zaczerpnięciu jeszcze jednego bajtu z bufora podglądu i sprawdzeniu, czy taka operacja nie przyczyni się do znalezienia dłuższego dopasowania.

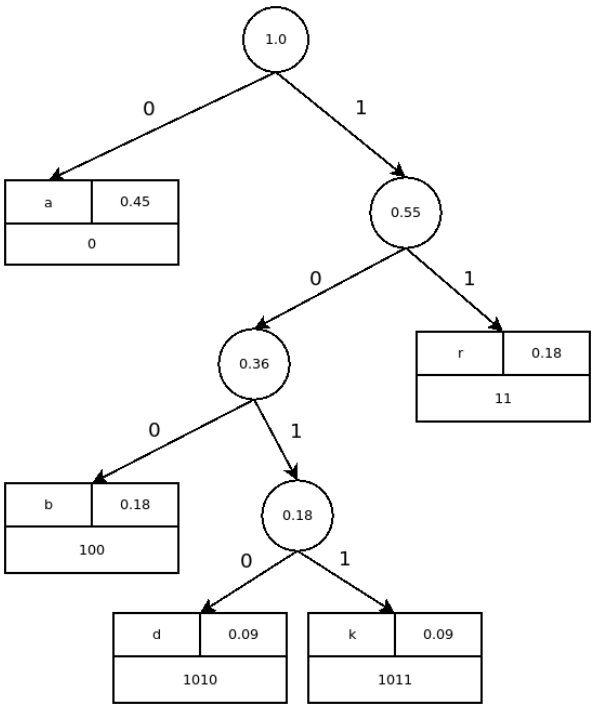
Tego typu sztuczki stosowane są w celu uzyskania większej wydajności: szybszej kompresji/dekompresji albo mniejszego rozmiaru pliku wynikowego. Więcej ciekawych szczegółów można znaleźć na stronie [10].

Istnieją specjalne narzędzia, takie jak lzbench [7], służące do mierzenia wydajności, a także specjalne referencyjne korpusy, takie jak Silesia compression corpus [8], do przeprowadzania badań. Wydajność różnych algorytmów, na podstawie repozytorium algorytmu kompresji Zstandard [9] przedstawiono w Tabeli 1.

Ciekawostka: wspomniany korpus użyty w badaniach pochodzi z polskiego podwórka, tak samo jak algorytm ANS [10] wykorzystywany w Zstandard.

Algorytm kompresji	Współczynnik kompresji	Szybkość kompresji	Szybkość dekompresji
zstd 1.4.4 -1	2.884	520 MB/s	1600 MB/s
zlib 1.2.11 -1	2.743	110 MB/s	440 MB/s
brotli 1.0.7 -0	2.701	430 MB/s	470 MB/s
quicklz 1.5.0 -1	2.238	600 MB/s	800 MB/s
lzo1x 2.09 -1	2.106	680 MB/s	950 MB/s
lz4 1.8.3	2.101	800 MB/s	4220 MB/s
snappy 1.1.4	2.073	580 MB/s	2020 MB/s
lzf 3.6 -1	2.077	440 MB/s	930 MB/s

Tabela 1. Ranking algorytmów kompresji (źródło: Zstandard [9])



Rysunek 3. Przykładowe drzewo Huffmana dla hasła „abrakadabra". Liście drzewa zawierają literki, prawdopodobieństwa ich wystąpienia oraz ich kodowanie binarne

Hasło „abrakadabra" po zakodowaniu przyjmuje zatem postać przedstawioną w Tabeli 3.

a	b	r	a	k	a	d	a	b	r	a
0	100	11	0	1011	0	1010	0	100	11	0

Tabela 3. Zapis binarny frazy „abrakadabra"

W zapisie ciągłym wygląda to następująco:

```
01001101 01101010 0100110_
```

Jak widać, udało się skompresować hasło „abrakadabra" do 3 bajtów i jest jeszcze 1 bit luzu!

INDEKSOWANIE ARCHIWUM GZIP

W poprzednich sekcjach uwaga była skupiona na tajnikach działania algorytmu DEFLATE. Dane traktowane są jako strumień, przez które przechodzi się pływającym okienkiem (buforem). Dekompresja działa odwrotnie do kompresji: bufor słownikowy jest odbudowywany w locie. Nasuwa się zatem pytanie wynikające z tytułu artykułu: w jaki sposób można dekompresować dane, rozpoczynając od losowego miejsca? Algorytm dekompresji opiera się przecież na odbudowywaniu bufora słownikowego ze skompresowanych danych!

Zanim zostanie opisany właściwy sposób, autor chciałby przypomnieć, że losowy dostęp do skompresowanych plików występuje w formatach takich jak ZIP. ZIP jest formatem archiwum, który zawiera w sobie kontenery. W każdym kontenerze znajduje się plik, który może być kompresowany indywidualnie. Jedną z opcji jest również kompresja za pomocą DEFLATE. Na końcu archiwum znajduje się katalog ze spisem kontenerów.

gzip + indexed_gzip jako kontener na pliki

Potrzeba jest matką wynalazków. W jednym z projektów autora pojawił się pewien techniczny problem związany z wydajnością tworzenia katalogów i plików, który doprowadził do wykorzystania techniki podobnej do tej przedstawionej w artykule. Aplikacja, w której problem wystąpił, jest rozproszona i pomaga inżynierom szybciej znajdować błędy w firmowym oprogramowaniu.

Jako wejście aplikacja przyjmuje zagnieżdżone archiwum, w którym łącznie występują tysiące plików i katalogów. Archiwa ważą średnio setki megabajtów i to mogłoby być spodziewanym źródłem problemów z wydajnością. Przyczyną problemów okazała się jednak mnogość plików, a nie ich rozmiar. Wszystko przez różnice w dostępie do dysków. Na lokalnym komputerze z dyskiem SSD operacje są szybkie, jednak w chmurze dyski twarde (przynajmniej w przypadku opisywanej aplikacji) są podpięte przez sieć, a sam system plików jest rozproszony (w przypadku opisywanej aplikacji jest to GlusterFS rozpięty na węzłach Kubernetesa i zarządzany przez Heketi).

Czas wykonania operacji na plikach w chmurze i na lokalnej maszynie może się znacząco różnić. Po przeprowadzeniu profilowania autor odkrył, że wywołanie `posix.mkdir`, czyli tworzenie katalogów, zajęło wysoką, trzecią lokatę w raporcie.

Aby wyobrazić sobie skalę problemu, można użyć polecenia tworzącego 10 tysięcy katalogów na lokalnym komputerze z dyskiem SSD i porównać to z tym, co dzieje się w chmurze. Na lokalnym komputerze wyniki mogą być podobne do przedstawionych poniżej:

```
$ /usr/bin/time mkdir $(seq 1 10000)
0.00user 0.14system 0:00.27elapsed 53%CPU
(0avgtext+0avgdata 2332maxresident)k 824inputs+0outputs
(0major+140minor)pagefaults 0swaps
```

W chmurze, korzystając z GlusterFS, wyniki są zdecydowanie gorsze: czas poniżej sekundy zmienia się na prawie 3 minuty. Wyniki te należy przyjąć z dystansem, ponieważ mogą zależeć od wielu czynników, takich jak np. obciążenie maszyn wirtualnych czy obciążenie klastra GlusterFS. Jednak różnica jest na tyle duża, że można śmiało stwierdzić, iż operacje dyskowe w takich architekturach chmurowych mogą stać się wąskim gardłem.

```
$ /usr/bin/time mkdir $(seq 1 10000)
0.04user 0.90system 2:43.88elapsed 0%CPU
(0avgtext+0avgdata 2624maxresident)k 0inputs+0outputs (0major+127minor)
pagefaults 0swaps
```

Widać też, że lokalnie procesor był wykorzystany przez 53% czasu, a w chmurze 0%. Może to wynikać z faktu, że dyski są sieciowe, a więc każda operacja blokuje się na sieci. Procesor zaś czas spędza w stanie bezczynności.

Aby rozwiązać ten problem, zaprzestano tworzenia wielu katalogów i małych plików. Zastąpione to zostało jednym dużym, skompresowanym plikiem GZIP z indeksem. Coś podobnego można by uzyskać, pakując wszystko w archiwum ZIP. Z dużą liczbą małych plików większość systemów plików radzi sobie podobnie – nie najlepiej.

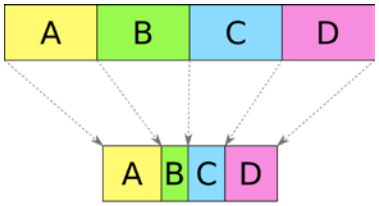
Zindeksowane pliki GZIP są bardziej skomplikowane, ale mają też swoje zalety:

- GZIP, w przeciwieństwie do ZIP, jest kompresją solidną [17], która w niektórych przypadkach uzyskuje lepszy współczynnik kompresji. W ramce „Kompresja solidna" zawarto wytłumaczenie, dlaczego się tak dzieje, wraz z przykładem.
- ZIP wymaga rozbicia danych na pliki. Nie zawsze jednak takie rozbicie jest praktyczne.

W dalszej części artykułu skupiono się na rozwiązaniu GZIP, jednak ostatnia sekcja jest uniwersalna i może znaleźć zastosowanie również dla formatu ZIP.

Indexed_gzip znajduje zastosowanie w formatach takich jak np. NIfTI-1 [13] (format danych wykorzystywany w neuroobrazowaniu) czy FASTQ [14] (wykorzystywany m.in. do sekwencji nukleotydów). Jednak do przykładu indeksowania zostały wybrane tradycyjne pliki, ponieważ są tworem bardziej znanym i nie wymagają dalszego wyjaśniania. Wykorzystane zostaną 4 pliki – A, B, C i D – o jednakowym rozmiarze, lecz różnej entropii. Przed kompresją pliki te są ze sobą połączone w jeden duży plik (bez użycia np. programu tar). Dla każdego z nich należy zapamiętywać przesunięcie (offset) oraz rozmiar.

Po kompresji rozmiar pliku najprawdopodobniej nie zostanie zachowany. Każdy z oryginalnych plików w swojej skompresowanej wersji zaczyna i kończy się w konkretnych miejscach. Mapowanie to zilustrowano na Rysunku 4.



Rysunek 4. Mapowanie przesunięć plików nieskompresowanych i skompresowanych

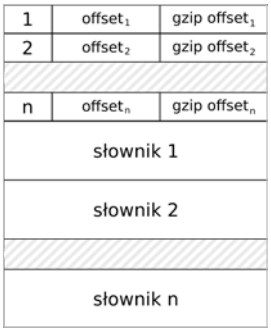
Podczas dekompresji całego pliku bufor słownikowy LZ77 jest odtwarzany od początku, tak jak przedstawiono na Rysunku 2. Zatem aby móc rozpocząć dekompresję w dowolnym miejscu, należy znać zawartość całego bufora słownikowego dla tego miejsca.

Można to uzyskać przez zapamiętywanie bufora słownikowego w każdym kroku, lecz taki sposób jest skrajnie niepraktyczny – zapamiętane dane przekroczyłyby rozmiar samego archiwum.

Praktycznym rozwiązaniem jest zapamiętanie zawartości bufora słownikowego tylko dla wybranych punktów w specjalnym indeksie. Punkty takie nazywamy punktami dostępu (ang. *entry points*). W przypadku jednorodnego rozmieszczenia punktów uzyskać można przewidywalny czas dekompresji, niezależnie od tego, który fragment został wybrany. Natomiast istotnym parametrem jest zagęszczenie tychże punktów. Należy je odpowiednio dobrać, najlepiej biorąc pod uwagę entropię danych oraz charakterystykę dostępu. Niestety nie zawsze ta wiedza jest dostępna w trakcie projektowania rozwiązania.

Na przykład – jeżeli kompresowane dane to nierozłączne fragmenty o rozmiarze około 50 MiB każdy, to tworzenie punktów dostępu co 1 MiB mija się z celem – taka precyzja nie jest potrzebna. Punkty dostępu rozmieszczone co 25 MiB powinny zapewnić porównywalną wydajność bez narzutu na rozmiar indeksu. Zbyt rzadkie rozmieszczenie punktów zadziała w drugą stronę: ilość danych, które należy zdekompresować, stanie się nieproporcjonalnie duża w stosunku do rozmiaru oryginalnych danych. W efekcie nie będzie dużej różnicy czasu pomiędzy dekompresją całego pliku a tak dużych fragmentów.

Indeks poza pomocniczymi informacjami o archiwum zawiera informacje o wszystkich punktach dostępu oraz zawartość buforów słownikowych dla każdego z nich. Zagęszczenie punktów dostępu wpływa bezpośrednio na rozmiar indeksu, również ten fizyczny na dysku. Utworzenie indeksu wymaga jednokrotnej dekompresji archiwum. Przykładową strukturę indeksu przedstawiono na Rysunku 5.



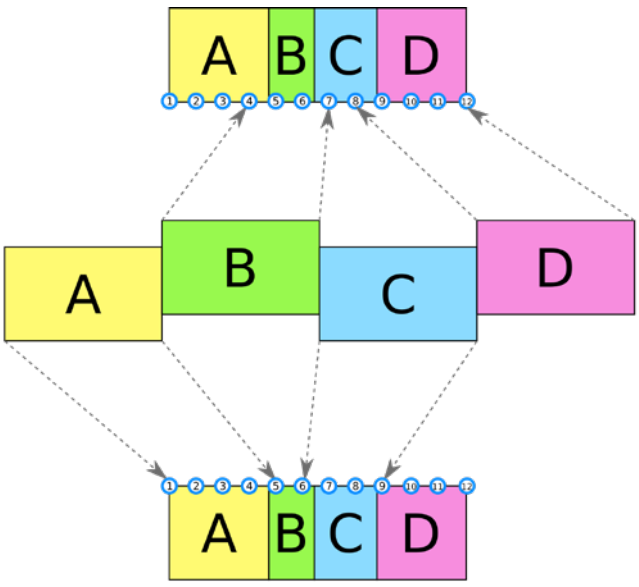
Rysunek 5. Ilustracja struktury indeksu. Rozmiary pól nie są proporcjonalne

Informacje zawarte w indeksie umożliwiają dekompresję wybranych fragmentów bez dekompresji całego pliku. Ponieważ punkty dostępu są ułożone równomiernie, rzadko kiedy początek interesującego fragmentu znajdzie się w tym samym miejscu co punkt dostępu. Zatem dekompresja najprawdopodobniej będzie się zaczynała przed początkiem wybranego fragmentu. Wszystkie zdekompresowane dane poprzedzające wybrany fragment są odrzucane.

Na Rysunku 6 przedstawiono wspomniane wcześniej w tekście przykładowe pliki z 12 punktami dostępu. Znając przesunięcia i rozmiary plików, można wyznaczyć dla każdego z plików:

» od którego punktu dostępu należy rozpocząć dekompresję,
» przed którym punktem dostępu dekompresja się zakończy.

Na przykład dla pliku B będą to punkty nr 4 i 7, a dla pliku C: 6 i 9.



Rysunek 6. Mapowanie przesunięć plików nieskompresowanych i skompresowanych zakładające punkty dostępu w każdej możliwej pozycji

Kompresja solidna

Często spotykanym scenariuszem jest kompresja wielu plików naraz. Kompresję taką można przeprowadzić na dwa sposoby: solidnie lub niesolidnie. Różnica polega na kolejności wykonywania zadań. W przypadku kompresji solidnej pliki najpierw są ze sobą łączone, a później kompresowane. Kompresja niesolidna kompresuje wszystkie pliki osobno, a dopiero później je łączy. Format ZIP wykorzystuje kompresję niesolidną. Formaty takie jak 7z, RAR czy GZIP (z wykorzystaniem np. programu tar) stosują kompresję solidną.

Obydwie metody kompresji mają swoje zalety i wady. Zaletą kompresji solidnej jest lepszy współczynnik kompresji w przypadku plików, które zawierają dużo powtórzeń tych samych danych. Dobrymi przykładami takich danych są np. przepisy kucharskie lub kod źródłowy. Listing 2 zawiera przykład porównujący kompresję przepisów z repozytorium GitHuba matkoniecz/Przepisy [16]. W przepisach słowa się często powtarzają. Kompresja niesolidna nie może jednak tego wykorzystać. W efekcie kompresja solidna produkuje plik, który jest o połowę mniejszy!

Listing 2. Porównanie kompresji solidnej i niesolidnej

```
# Wszystkie przepisy zątczone: około 124 kB
$ cat **/*.txt | wc -c
126589
# Kompresja solidna: około 45 kB
$ cat **/*.txt | gzip -c | wc -c
45874
# Kompresja niesolidna: około 104 kB
$ zip -q - **/*.txt | wc -c
106409
```

Biblioteka indexed_gzip implementuje tworzenie, eksportowanie oraz importowanie indeksu oraz pozwala na operowanie na archiwum GZIP tak, jakby był to zwyczajny plik. Podczas dostępu do pliku „pod spodem" dekompresowane są tylko wymagane fragmenty archiwum zgodnie z powyższym opisem.

IMPLEMENTACJA

W kolejnych sekcjach artykułu przedstawiono przykładową implementację wykorzystującą indexed_gzip w wersji 0.8.10. Kod był testowany na komputerze z zainstalowanym systemem operacyjnym GNU/Linux i Pythonem 3.7.4. Biblioteka wspiera również system Windows, jednak poniższa implementacja nie była na tym systemie testowana.

Uwaga! Zawarty kod nie jest kodem produkcyjnym! Jego konstrukcja została dobrana względem artykułu. Wiele kwestii, takich jak styl, bezpieczeństwo, rozszerzalność, testowalność itp. zostało pominiętych!

Kompresja

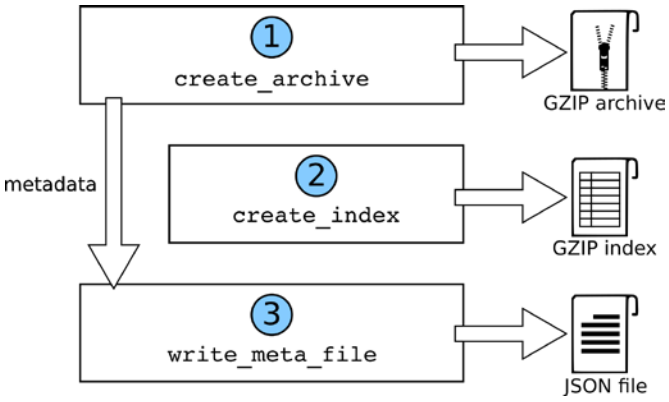
Przedstawiona implementacja będzie kompresować wiele plików o różnym rozmiarze w jedno archiwum GZIP. Dla każdego dodawanego pliku zapamiętane zostanie jego przesunięcie względem początku. Następnie tworzony będzie indeks za pomocą biblioteki indexed_gzip. Wszystkie zapamiętane przesunięcia zapisane zostaną w pomocniczym pliku z metadanymi. Następnie zestaw tych trzech plików wykorzystany będzie w celu dekompresji jednego z oryginalnych plików bez dekompresji całego archiwum.

Zawartość kompresowanych plików nie ma znaczenia z punktu widzenia samej implementacji (ale oczywiście wpłynie na jakość kompresji). W Listingu 3 przedstawiono prosty skrypt powłoki, który utworzy zadaną liczbę plików z losowymi danymi i o losowym rozmiarze pomiędzy 1 a 25 MiB.

Listing 3. Skrypt powłoki tworzący pliki z losową zawartością (create-random-files)

```
#!/usr/bin/env bash
set -euo pipefail

if ! [[ $# -eq 2 ]]; then
  >&2 echo "usage: $0 output_directory num_files"
  exit 1
fi

out_dir=$1
num_files=$2
rm -r $out_dir && mkdir -p $out_dir

for i in $(seq 1 $num_files); do
    random_size=$(((RANDOM % 25) + 1))
    dd                                \
      if=/dev/urandom                 \
      bs=1M                           \
      count=$random_size              \
      of="${out_dir}/file${i}.dat" \
      &>/dev/null
done
```

W Listingu 4 przedstawiono kod związany z kompresją utworzonych plików, utworzeniem indeksu oraz pliku z metadanymi. Odpowiadają za to odpowiednio metody `create_archive`, `create_index` i `write_meta_file`. Na Rysunku 7 zilustrowano kolejność operacji i wykorzystywane dane.



Rysunek 7. Diagram przedstawiający kroki tworzenia plików

Listing 4. Skrypt tworzący skompresowany kontener wraz z indeksem i plikiem meta (compress.py)

```
#!/usr/bin/env python3

import argparse, gzip, pathlib, hashlib, json, typing
import indexed_gzip as igz

def copyfileobj_with_md5(fsrc, fdst, length=16 * 1024):
    """
    Funkcja kopiująca dane pomiędzy dwoma obiektami
    plikowymi - skopiowana z shutil.copyfileobj oraz
    wzbogacona o zwracanie liczby skopiowanych bajtów
    oraz sumę md5.
    """
    num_copied_bytes = 0
    md5_sum = hashlib.md5()
    while 1:
        buf = fsrc.read(length)
        if not buf:
            break
        fdst.write(buf)
        num_copied_bytes += len(buf)
        md5_sum.update(buf)
    return (num_copied_bytes, md5_sum)

class IndexedGzip(typing.NamedTuple):
    input_dir: pathlib.Path
    output_dir: pathlib.Path
    pattern: str

    @property
    def container_path(self):
        return str(self.output_dir / "container.gz")

    @property
    def index_path(self):
        return self.container_path + ".index"

    @property
    def meta_path(self):
        return self.container_path + ".json"

    def create_archive(self):
        """
        Tworzenie archiwum GZIP na podstawie wszystkich
        plików pasujących do wzorca GLOB znajdujących
        się w katalogu wejściowym.
        """
        meta = {}
        offset = 0

        with gzip.open(self.container_path, "wb") as f_out:
            for path in self.input_dir.glob(self.pattern):
                with path.open("rb") as f_in:
                    size, md5_sum = \
                        copyfileobj_with_md5(f_in, f_out)
                    meta[path.name] = {
                        "offset": offset,
                        "size": size,
                        "md5": md5_sum.hexdigest()
                    }
                    offset += size

        return meta
```

```
    def create_index(self):
        """
        Tworzenie indeksu GZIP za pomocą biblioteki
        indexed_gzip. Rozmieszczenie punktów dostępu
        co 1 MiB.
        """
        igzf = igz.IndexedGzipFile(
                self.container_path,
                spacing=1024**2)
        igzf.build_full_index()
        igzf.export_index(self.index_path)
        igzf.close()

    def write_meta_file(self, meta_obj):
        """
        Zapisanie na dysk metadanych w formacie json.
        """
        with open(self.meta_path, "w") as f_out:
            json.dump(meta_obj, f_out, indent=2)

    def create(self):
        """
        Główna metoda tworząca archiwum, indeks oraz
        plik z metadanymi.
        """
        meta = self.create_archive()
        self.create_index()
        self.write_meta_file(meta)

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("input_dir")
    ap.add_argument("output_dir")
    args = ap.parse_args()

    input_dir = pathlib.Path(args.input_dir)
    output_dir = pathlib.Path(args.output_dir)

    IndexedGzip(input_dir, output_dir, "*.dat").create()
```

W metodzie `create_archive` za pomocą gzip.open otwierany jest kompresowany „w locie" plik wyjściowy. Następnie wszystkie pliki z katalogu wejściowego, które pasują do wzorca, są dodawane do archiwum GZIP.

W przedstawionym kodzie źródłowym użyta została modyfikacja funkcji `copyfileobj` z wbudowanej biblioteki shutil: `copyfileobj_with_md5`. Ta biblioteczna funkcja kopiuje dane z jednego obiektu plikowego do drugiego bez ładowania całości danych do pamięci głównej. W przypadku małych plików nie ma to wielkiego znaczenia, jednak jeżeli przetwarzane pliki są duże, istotne jest, aby zużycie pamięci było stałe. Wprowadzone zmiany polegają na zliczeniu skopiowanych bajtów oraz wyliczeniu sumy md5. Zliczanie bajtów przydaje się do obliczenia przesunięcia następujących po sobie plików (zmienna `offset`). Suma md5 pozwoli zweryfikować integralność danych. Informacje te zapisywane są dla każdego pliku w słowniku meta.

pathlib – wygodna praca z systemami plików w Pythonie

W Listingu 4 użyta jest wbudowana od wersji 3.4 Pythona biblioteka pathlib, dzięki której praca z plikami i katalogami jest łatwiejsza w porównaniu ze sposobem tradycyjnym:

- wygodne `input_dir.glob` zamiast biblioteki glob lub `os.listdir`
- `path.open` zamiast `open()` z przekazanym napisem jako argument
- `path.name` zamiast `os.path.basename()`

Dekompresja

Dekompresja pliku przy użyciu indexed_gzip jest jeszcze prostsza niż kompresja, co zilustrowano w Listingu 5.

Listing 5. Skrypt dekompresujący pliki (decompress.py)

```
#!/usr/bin/env python3

import argparse, json, pathlib, sys
import indexed_gzip as igz

def decompress(input_dir, filename):
    container_path = str(input_dir / 'container.gz')
    index_path = container_path + '.index'
    meta_path = container_path + '.json'

    with open(meta_path) as f:
        file_meta = json.load(f)[filename]

    with igz.IndexedGzipFile(container_path) as igzf:
        igzf.import_index(index_path)
        igzf.seek(file_meta['offset'])
        sys.stdout.buffer.write(igzf.read(file_meta['size']))

if __name__ == '__main__':
    ap = argparse.ArgumentParser()
    ap.add_argument('input_dir')
    ap.add_argument('filename')
    args = ap.parse_args()
    decompress(pathlib.Path(args.input_dir), args.filename)
```

Dekompresja odbywa się w funkcji `decompress`, gdzie najpierw na podstawie pliku meta wyłuskiwane są informacje o pliku, następnie otwierane jest archiwum GZIP i importowany utworzony uprzednio indeks. Po załadowaniu indeksu, metodą `IndexedGzipFile.seek` można się przesunąć w odpowiednie miejsce tak, jak gdyby to był nieskompresowany plik, i odczytać zadaną liczbę bajtów z pomocą `IndexedGzipFile.read`. Odczytane dane w przedstawionym przykładzie kopiowane są bezpośrednio do bufora wyjścia standardowego. Ułatwi to przekazanie wyjścia za pomocą potoków (ang. *pipe*) do następnych programów.

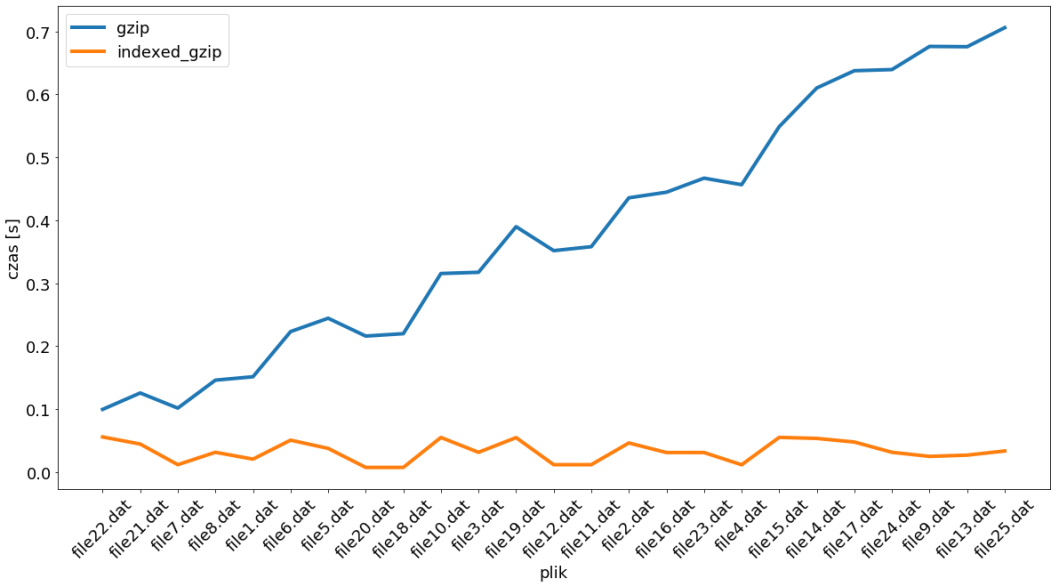
Warto wspomnieć, że ręczne ładowanie indeksu nie jest wymagane. Jeżeli nie zostanie to zrobione przez programistę, to przy pierwszej operacji biblioteka leniwie go utworzy. Należy jednak pamiętać o tym, że podczas tworzenia indeksu następuje dekompresja całego pliku!

POPRAWNOŚĆ I WYDAJNOŚĆ

Zawsze warto organoleptycznie potwierdzić, że po uruchomieniu wszystko działa jak należy:

Listing 6. Weryfikacja poprawności działania rozwiązania

```
$ # Tworzenie 25 losowych plików
$ ./create-random-files input 25

$ # Sprawdzenie rozmiarów i md5 3 losowych plików
$ stat --format="%n: %s" input/* | head -n 3
input/file10.dat: 25165824
input/file11.dat: 4194304
input/file12.dat: 4194304
$ find input -type f | head -n 3 | xargs md5sum
942019a4b6c03708c7bbb81384cb902f  input/file22.dat
ed28f6f21c5a4dc9e007f068f743e680  input/file21.dat
7f1108463591f1c14e60f90d6c0f59ff  input/file7.dat

$ # Tworzenie indeksu i pliku meta
$ mkdir indexed && ./compress.py input indexed/
$ stat --format="%n: %s" indexed/*
indexed/container.gz: 360820207
indexed/container.gz.index: 11278092
indexed/container.gz.json: 2872

$ # Dekompresja wybranych plików i weryfikacja ich sum md5
$ for N in 22 21 7; do
for> ./decompress.py indexed/ "file${N}.dat" | md5sum
for> done
942019a4b6c03708c7bbb81384cb902f  -
ed28f6f21c5a4dc9e007f068f743e680  -
7f1108463591f1c14e60f90d6c0f59ff  -
```



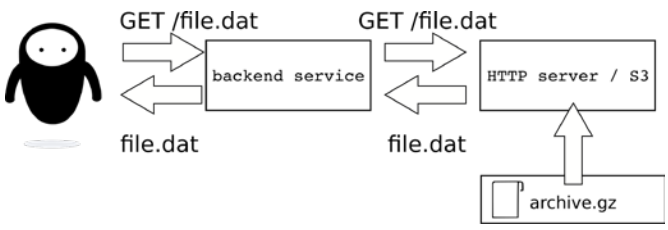
Rysunek 8. Porównanie wydajności GZIP i indexed_gzip

Poza poprawnością warto również sprawdzić wydajność. Wyniki, wydawałoby się, oczywiste, jednak praktyka pokazuje, że intuicja nie jest dobrym doradcą w sprawach wydajności. Na Rysunku 8 przedstawiono czasy dekompresji zmierzone dla dwóch rozwiązań, z czego jedno wykorzystuje indexed_gzip, a drugie nie. Na osi OX rozmieszczone są pliki, które były kompresowane w kolejności, w jakiej znalazły się w kontenerze. Wyraźnie widać, że wydajność tradycyjnego rozwiązania przy użyciu GZIP rośnie liniowo, podczas gdy użycie indexed_gzip niweluje ten wzrost kosztem przetrzymywania indeksu na dysku. Rozmiar pliku z indeksem zależy od liczby punktów dostępu i charakterystyki danych. Dla przedstawionego w artykule przykładu jest to około 11 megabajtów (Listing 6).

INTEGRACJA Z S3/HTTP

Co prawda indeksowane pliki GZIP (lub ZIP) mają wiele zastosowań, wykorzystanie ich np. w aplikacjach webowych może przynieść korzyści w postaci mniejszego zużycia procesora kosztem niewielkiej ilości przestrzeni dyskowej. Z tego z kolei wynika mniejsze zużycie prądu oraz mniejsze średnie opóźnienia u klientów.

Na Rysunku 9 przedstawiono jeden ze scenariuszy, w których można wykorzystać indexed_gzip po stronie serwera: na S3/HTTP przetrzymywane są duże archiwa GZIP. Pomimo że klient jest zainteresowany tylko pojedynczymi plikami z archiwum, usługa backend i tak musi pobrać je całe w celu dekompresji i wycięcia wybranego fragmentu.



Rysunek 9. Scenariusz pobierania pojedynczego pliku z archiwum bez indexed_gzip

Poprzez wprowadzenie indexed_gzip można m.in.:

- zredukować ruch sieciowy pomiędzy serwerami S3/HTTP,
- zmniejszyć zużycie CPU we wszystkich usługach biorących udział w spełnieniu żądania klienta.

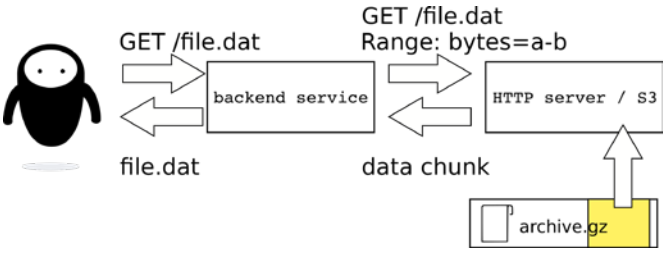
Jednak pojawia się pytanie – w jaki sposób zdekompresować w usłudze backend jedynie wycinek archiwum? Okazuje się, że sprytne wykorzystanie wywołania systemowego `ftruncate`, nagłówka HTTP Range oraz drobna zmiana w bibliotece indexed_gzip to umożliwiają.

Trik polega na tym, aby sztucznie tworzyć za pomocą `ftruncate` tzw. rozsiany plik (ang. *sparse file*) o takim samym rozmiarze jak archiwum i wklejać do niego:

- nagłówek archiwum na samym początku, dzięki czemu plik będzie owo archiwum „udawał". W nagłówku znajdują się istotne informacje, bez których rozwiązanie nie zadziała,
- wycinek archiwum zawierający w sobie żądany plik. Do wyznaczenia tego regionu należy posłużyć się indeksem.

Tak spreparowany rozsiany plik zajmuje na dysku tylko tyle miejsca, ile zajmuje nagłówek i wycinek. Aby z serwera S3/HTTP pobrać wycinek pliku, należy wykorzystać nagłówek HTTP Range.

Na Rysunku 10 przedstawiono to samo rozwiązanie wykorzystujące indexed_gzip.



Rysunek 10. Scenariusz pobierania pojedynczego pliku z archiwum z indexed_gzip

Warto przed implementacją takiego rozwiązania upewnić się, że wykorzystywane systemy plików obsługują pliki rozsiane. Niektóre takiego wsparcia nie oferują!

Niestety, biblioteka indexed_gzip nie udostępnia możliwości bezpośredniego odczytywania punktów dostępu, co jest wymagane, aby przedstawiane rozwiązanie było możliwe. Można to jednak w prosty sposób obejść, modyfikując ją (licencja na to pozwala). Listing 7 zawiera zmiany, jakie należy zaaplikować, aby punkty dostępu stały się widoczne z poziomu Pythona.

Listing 7. Zarys zmian wymaganych do uzyskania dostępu do punktów zapisanych w indeksie (źródło [15])

```
diff --git a/.../indexed_gzip.pyx b/.../indexed_gzip.pyx
index fdc676d..a1d7d40 100644
--- a/indexed_gzip/indexed_gzip.pyx
+++ b/indexed_gzip/indexed_gzip.pyx
@@ -288,6 +288,10 @@ cdef class _IndexedGzipFile:

       return proxy()

+  def points(self):
+      for i in range(self.index.npoints):
+          point = self.index.list[i]
+          yield (point.uncmp_offset, point.cmp_offset)

  def fileno(self):
      """Calls ``fileno`` on the underlying file object.
@@ -855,6 +859,7 @@class IndexedGzipFile(io.BufferedReader):
   self.export_index     = fobj.export_index
   self.fileobj          = fobj.fileobj
   self.drop_handles     = fobj.drop_handles
+  self.points           = fobj.points

   super(IndexedGzipFile, self).__init__(fobj, buffer_size)
```

HTTP Range

Protokół HTTP wspiera wiele nagłówków. Jednym z nich jest `Range:`, który pozwala na pobranie z serwera (jeżeli serwer obsługuje takie zapytania) wycinku odpytywanego zasobu. Jest to przydatne, gdy klient chce wznowić pobieranie lub nie potrzebuje całej zawartości pliku (przypadek z artykułu).

```
$ export URL="https://slawomir.net/hello.txt"
$ curl $URL
HELLO WORLD!
tu Sławomir!
journey AWAITS!

$ curl -H "Range: bytes=0-11" $URL
HELLO WORLD!
```

Serwer może też być odpytany o wiele zakresów naraz. W takim przypadku odpowiedź zostanie odpowiednio podzielona (ang. *multipart*).

```
curl -H "Range: bytes=0-3,35-41" $URL

--28e92328b2fbedc1
Content-type: text/plain
Content-range: bytes 0-3/43

HELL
--28e92328b2fbedc1
Content-type: text/plain
Content-range: bytes 35-41/43

AWAITS!
--28e92328b2fbedc1--
```

magiczny ftruncate

`{,f}truncate{,64}` to wywołania systemowe (ang. *syscall*) systemów operacyjnych zgodnych ze standardem POSIX (m.in. GNU/Linux), które rozszerzają lub skracają plik do określonego w argumencie rozmiaru. Pozwalają one na szybkie tworzenie dużych rozsianych plików wypełnionych zerami, jeżeli tylko takie pliki są wspierane przez użyty system plików.

Nie wszystkie systemy plików wspierają pliki rozsiane. W przypadku kiedy dany system plików nie ma takiego wsparcia, cała przestrzeń zostanie wypełniona zerami. Jeżeli jednak wsparcie jest, to można tworzyć wielkie pliki, a fizyczne miejsce na nośniku będą zajmowały tylko te dane, które do pliku zostały faktycznie zapisane.

Można by się pokusić nawet o dekompresję po stronie klienta, jeżeli dane z sąsiadujących plików w pliku GZIP nie zawierają wrażliwych informacji. Ten temat jednak wykracza poza ramy artykułu.

FIN

Wielu programistów żyje w ciągłym sprincie: pojawiają się coraz to nowe języki, biblioteki, procesy i frameworki. Aby utrzymać się na powierzchni, należy nieustannie się doszkalać. Warto jednak w tej niekończącej się nauce na warsztat wziąć nie nowinkę z HackerNews (lub odpowiednika), lecz fundamentalne rozwiązania, na których bazuje bardzo dużo rozwiązań pochodnych. Algorytm DEFLATE jest jednym z nich.

Zaglądając do środka, odkrywa się, że idee stojące za skomplikowaną implementacją są proste do zrozumienia. Co więcej, znając te szczegóły, można wykorzystywać takie technologie w sposób nieoczywisty, co może okazać się niezłym pomysłem na przykład na optymalizację oprogramowania.

Podziękowania: Damian „kele" Bogel

W sieci

[0] https://github.com/madler/zlib/blob/master/examples/zran.c
[1] https://github.com/pauldmccarthy/indexed_gzip
[2] https://code.google.com/archive/p/jzran/
[3] https://tinyurl.com/yclva46f
[4] https://tools.ietf.org/html/rfc1951
[5] https://en.wikipedia.org/wiki/Longest_common_substring_problem
[6] https://github.com/madler/zlib/blob/master/doc/algorithm.txt
[7] https://github.com/inikep/lzbench
[8] http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia
[9] https://github.com/facebook/zstd
[10] https://www.euccas.me/zlib/
[11] https://en.wikipedia.org/wiki/Morse_code
[12] https://en.wikipedia.org/wiki/Shannon%27s_source_coding_theorem
[13] https://nifti.nimh.nih.gov/
[14] https://en.wikipedia.org/wiki/FASTQ_format
[15] https://github.com/szborows/indexed_gzip/commit/ee3c146
[16] https://github.com/matkoniecz/Przepisy
[17] https://en.wikipedia.org/wiki/Solid_compression

SŁAWOMIR ZBOROWSKI

https://slawomir.net

Programista z pasją, który pamięta czasy Linuxa łupanego. Pracuje jako architekt R&D we wrocławskim oddziale Nokii, gdzie zajmuje się rozwijaniem aplikacji rozproszonych. Po godzinach w wolnym czasie rozwija swoje miniprojekty.