Systemy archiwizacji danych - znajdowanie duplikatów

Kornel Jakubczyk

15.04.2009

Przewidywane zastosowanie

- Klient dostarcza strumień danych do zapisania

Przewidywane zastosowanie

- Klient dostarcza strumień danych do zapisania
- Do systemu zapisywane będą jego kolejne wersje, prawdopodobnie dość podobne

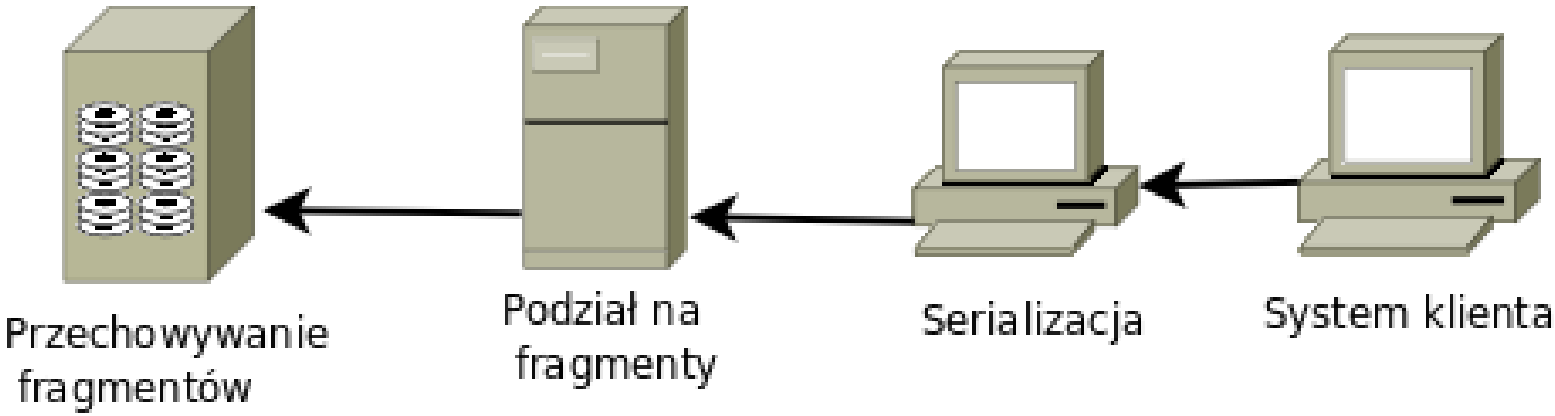
Przewidywane zastosowanie

- Klient dostarcza strumień danych do zapisania
- Do systemu zapisywane będą jego kolejne wersje, prawdopodobnie dość podobne
- Danych będzie dużo - chcemy minimalizować ich ilość

Przewidywane zastosowanie

- Klient dostarcza strumień danych do zapisania
- Do systemu zapisywane będą jego kolejne wersje, prawdopodobnie dość podobne
- Danych będzie dużo - chcemy minimalizować ich ilość
- Klient udostępnia nam pewne okienko czasowe na wykonanie kopii zapasowej - musimy zdążyć

Architektura systemu



Funkcja haszująca

- Dla każdego fragmentu (ang. chunk) wyliczamy wartość funkcji skrótu.

Funkcja haszująca

- Dla każdego fragmentu (ang. chunk) wyliczamy wartość funkcji skrótu.
- Można używać różnych np. MD5, poniżej zawsze używana będzie SHA1, generuje klucz długości 20 bajtów.

Funkcja haszująca

- Dla każdego fragmentu (ang. chunk) wyliczamy wartość funkcji skrótu.
- Można używać różnych np. MD5, poniżej zawsze używana będzie SHA1, generuje klucz długości 20 bajtów.
- Wyliczony klucz będzie jednoznacznie identyfikował fragment danych

Funkcja haszująca

- Dla każdego fragmentu (ang. chunk) wyliczamy wartość funkcji skrótu.
- Można używać różnych np. MD5, poniżej zawsze używana będzie SHA1, generuje klucz długości 20 bajtów.
- Wyliczony klucz będzie jednoznacznie identyfikował fragment danych
- Prawdopodobieństwo kolizji jest mniejsze od szansy na awarię dysku, można je zmniejszać stosując funkcje haszujące z dłuższym kluczem.

Chunk store

- Zapewne będzie to system rozproszony

Chunk store

- Zapewne będzie to system rozproszony
- Zapewniać będzie takie usługi jak bezpieczeństwo przez replikację, load balancing etc.

Chunk store

- Zapewne będzie to system rozproszony
- Zapewniać będzie takie usługi jak bezpieczeństwo przez replikację, load balancing etc.
- Adresowany zawartością - do znalezienia bloku powinna wystarczyć sama wartość jego funkcji haszującej.

Chunk store

- Zapewne będzie to system rozproszony
- Zapewniać będzie takie usługi jak bezpieczeństwo przez replikację, load balancing etc.
- Adresowany zawartością - do znalezienia bloku powinna wystarczyć sama wartość jego funkcji haszującej.
- Nie będziemy go tu omawiać

Współczynnik deduplikacji

- Stosunek wielkości oryginalnych danych do zużytego miejsca.
- Dążymy do jego maksymalizacji
- Bardzo zależny od samych danych

Rozmiar fragmentu

- Rozmiar fragmentu wpływa na współczynnik deduplikacji.
- Mniejsze fragmenty częściej się powtarzają

Rozmiar fragmentu

- Rozmiar fragmentu wpływa na współczynnik deduplikacji.
- Mniejsze fragmenty częściej się powtarzają
- Dłuższych fragmentów będzie mniej

Ilość metadanych

- Jest ważna w połączeniu z ilością fragmentów, wpływa na faktyczne zużycie przestrzeni w chunk store.

Ilość metadanych

- Jest ważna w połączeniu z ilością fragmentów, wpływa na faktyczne zużycie przestrzeni w chunk store.
- Zależy od konkretnej implementacji i zastosowania, będziemy zakładać ok 4-12 bajtów na fragment.

Tryb wsadowy

Zachowujemy dane na boku aby przetworzyć je później
Zalety - możliwość zastosowania wolniejszych algorytmów

Wady?

- Dodatkowego miejsca może być istotnie dużo

Wady?

- Dodatkowego miejsca może być istotnie dużo
- Nie oszczędzimy nic na przesyłaniu przez sieć

Wady?

- Dodatkowego miejsca może być istotnie dużo
- Nie oszczędzimy nic na przesyłaniu przez sieć
- Trudniejsza realizacja - Wszystkie usługi związane np. z bezpieczeństwem trzeba implementować osobno dla tymczasowego i właściwego miejsca przechowywania.

Wady?

- Dodatkowego miejsca może być istotnie dużo
- Nie oszczędzimy nic na przesyłaniu przez sieć
- Trudniejsza realizacja - Wszystkie usługi związane np. z bezpieczeństwem trzeba implementować osobno dla tymczasowego i właściwego miejsca przechowywania.
- W związku z tym zajmiemy się tylko podziałem w locie.

FSP - Fixed Sized-partitioning

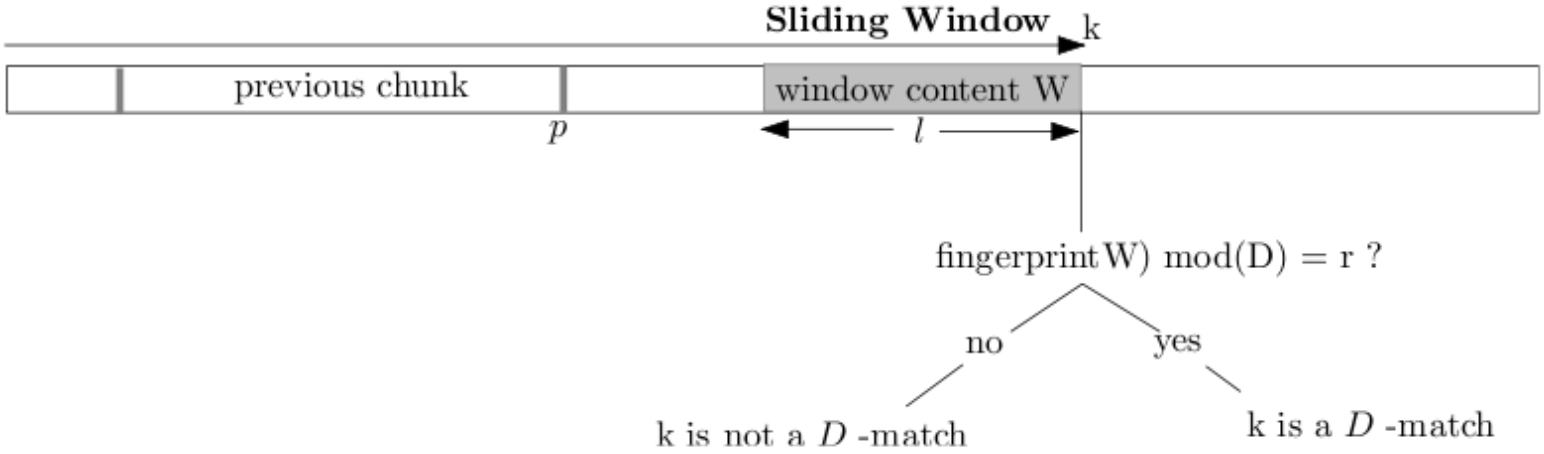
Dzielimy strumień danych jak wypadnie
Zalety:

- prosty
- być może łatwiejsza budowa chunk store
- mało metadanych
- długość bloku danych może być dostosowana do wielkości bloku dyskowego

FSP - Fixed Sized-partitioning

Wadą jest kiepska deduplikacja, czyli to co chcemy osiągnąć
Wystarczy wstawić 1 bajt na początku strumienia
Potwierdza się to też w wynikach eksperymentów

Przesuwane okno



[1]

Długość generowanego fragmentu

- Zakładamy losowość danych wejściowych i dobre własności funkcji haszującej
- Długość okna to typowo od 32 do 96 bitów
- Ustalamy, że pewna ilość ostatnich bitów “odcisku palca” powinna się równać jakiejś ustalonej wartości

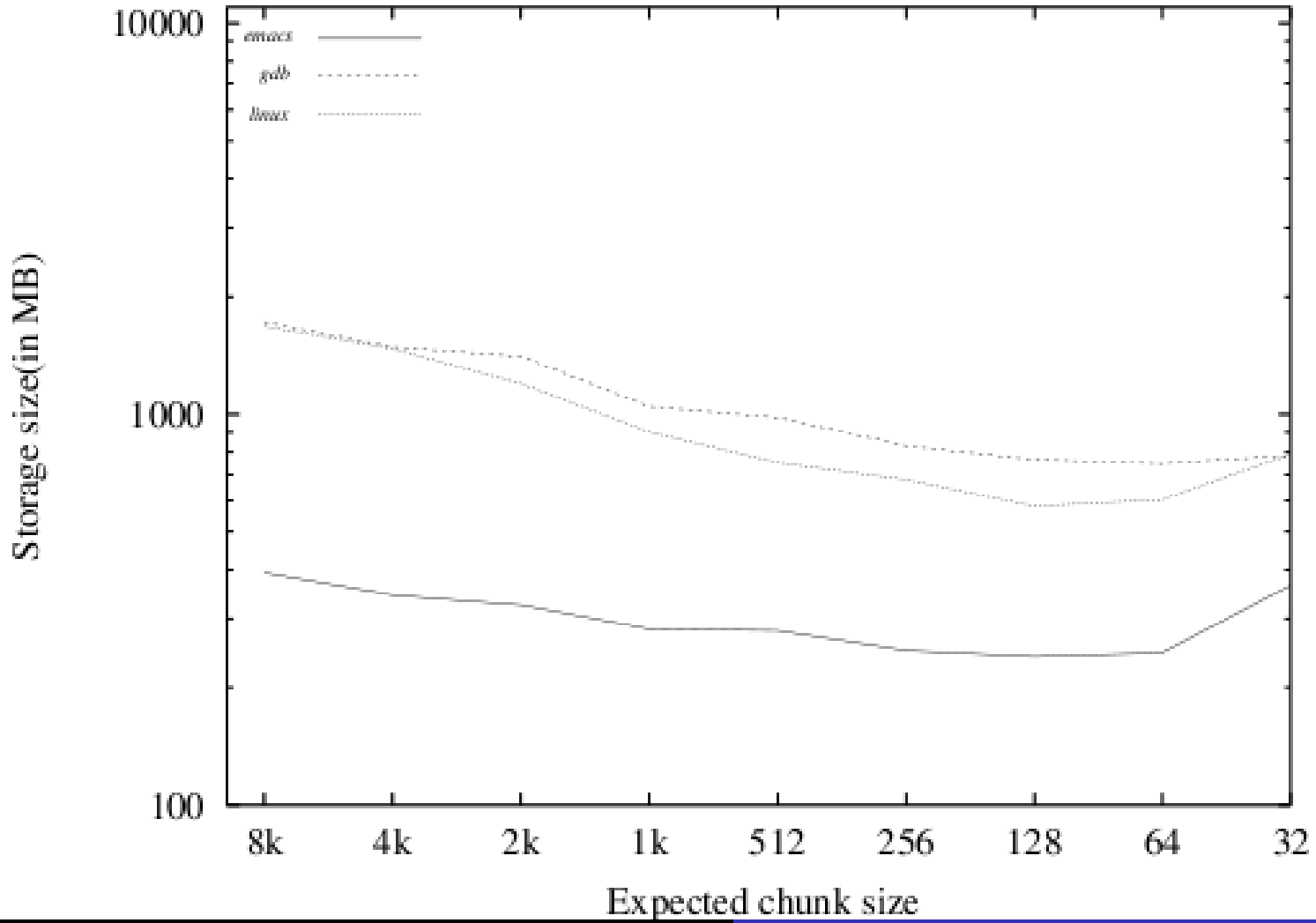
Długość generowanego fragmentu

- Zakładamy losowość danych wejściowych i dobre własności funkcji haszującej
- Długość okna to typowo od 32 do 96 bitów
- Ustalamy, że pewna ilość ostatnich bitów “odcisku palca” powinna się równać jakiejś ustalonej wartości
- Jeśli np. 13 to otrzymamy fragmenty długości ok. 8kB

Długość generowanego fragmentu

- Zakładamy losowość danych wejściowych i dobre własności funkcji haszującej
- Długość okna to typowo od 32 do 96 bitów
- Ustalamy, że pewna ilość ostatnich bitów “odcisku palca” powinna się równać jakiejś ustalonej wartości
- Jeśli np. 13 to otrzymamy fragmenty długości ok. 8kB
- Czy prawdziwe dane też są dostatecznie losowe?

Długość fragmentu



[3]

Rabin fingerprint

Jest tutaj typowo stosowany

- Traktujemy ciąg bitów jako współczynniki wielomianu nad Z_2
- Wybieramy pewien nierozkładalny wielomian P
- Wartością jest reszta z dzielenia wielomianu z wejścia przez P

Rabin fingerprint

Można go wyliczyć efektywnie używając operacji xor i przesunięcia bitowego.
Obliczenie jest w czasie linowym.

$$f(b_1,\cdots b_l) = r_k t^{k-1} + \cdots + r_2 t + r_1$$

$$f(b_1,\cdots b_l, b_{l+1}) = r_{k-1} t^{k-1} + \cdots + r_1 t + r_k (t^k \bmod P(t))$$

[2]

Znaczenie wariancji długości

- Załóżmy, że zapisujemy drugą wersję pliku podzielonego CDC, w której nastąpiła niewielka zmiana
- Można obliczyć [1], że wtedy ilość zapisanych “niepotrzebnie“ informacji jest proporcjonalna do ilości fragmentów dotkniętych zmianą oraz do kwadratu wariancji długości bloku
- Można próbować zmniejszać właśnie tą wariancję

Znaczenie wariancji długości

- Załóżmy, że zapisujemy drugą wersję pliku podzielonego CDC, w której nastąpiła niewielka zmiana
- Można obliczyć [1], że wtedy ilość zapisanych "niepotrzebnie" informacji jest proporcjonalna do ilości fragmentów dotkniętych zmianą oraz do kwadratu wariancji długości bloku
- Można próbować zmniejszać właśnie tą wariancję
- ale zmniejszanie wariancji do zera zaprowadzi znów do FSP

Intuicja

Jeśli zmiana występuje w losowym miejscu to jest bardziej prawdopodobne, że miejsce to należeć będzie do dłuższego fragmentu
czyli dłuższe fragmenty będziemy musieli zapisywać częściej, bo dotknie ich więcej zmian
ale zmiana jest mała czyli zapiszemy więcej niepotrzebnych danych z całego fragmentu
Potrzeba więc zmniejszać jednocześnie ilość zmienionych fragmentów jak i wariancję ich długości.

Poprawki

- Sklejanie małych fragmentów (SCM) - ignorujemy zbyt wczesne dopasowania
- czyli takie które wystąpią przed pewną progową długością L

Poprawki

- Sklejanie małych fragmentów (SCM) - ignorujemy zbyt wczesne dopasowania
- czyli takie które wystąpią przed pewną progową długością L
- Rozbijanie dużych bloków (BFS)- bloki dłuższe niż T dzielimy na fragmenty o stałej długości T
- (TD) dłuższe bloki dzielimy tak jak w CDC

Poprawki

- Sklejanie małych fragmentów (SCM) - ignorujemy zbyt wczesne dopasowania
- czyli takie które wystąpią przed pewną progową długością L
- Rozbijanie dużych bloków (BFS)- bloki dłuższe niż T dzielimy na fragmenty o stałej długości T
- (TD) dłuższe bloki dzielimy tak jak w CDC
- Można to wszystko połączyć i nieco poprawić

Algorytm Two Thresholds, Two Divisors

Wybieramy dwa dzielniki D i D', fragment odcinamy jeśli wartość odcisku palca przystaje do -1 modulo D

- Dopasowań szukamy dopiero kiedy mamy dostatecznie dużo danych dla bloku
- Jeśli znajdziemy dopasowanie względem D ucinamy blok
- Jeśli nie i dotarliśmy już do maksymalnej długości bloku sprawdzamy czy wcześniej były dopasowania do D'.
  - tak - ucinamy blok w odpowiednim miejscu
  - nie - ucinamy blok o maksymalnej dozwolonej długości

Porównanie

Algorithm	α_{rand}	α_{real}
BSW	2.04	2.44
BFS	1.98	2.07
TD	1.77	1.79
SCM	1.51	1.73
TTTD	1.51	1.52

[1]

Wartość alfa to współczynnik, określający stosunek długości niepotrzebnie zapisanych danych do średniej długości bloku.

Wniosek

Ogólnie te metody zachowują się gorzej dla rzeczywistych plików niż dla losowych danych.
Przyczyną jest występowanie powtarzających się części w tych plikach.

Motywacja

- Konflikt między rozmiarem bloku a ilością bloków postaramy się rozwiązać w inny sposób.

Motywacja

- Konflikt między rozmiarem bloku a ilością bloków postaramy się rozwiązać w inny sposób.
- Będziemy się starać zamknąć zmiany w krótkich nowych blokach, a stare pozostawiać niezmienione i długie

Motywacja

- Konflikt między rozmiarem bloku a ilością bloków postaramy się rozwiązać w inny sposób.
- Będziemy się starać zamknąć zmiany w krótkich nowych blokach, a stare pozostawiać niezmienione i długie
- Rezygnujemy z bezstanowości

Pomysł

- Wykorzystamy CDC do dzielenia strumienia na mniejsze podbloki, które potem będziemy łączyć w miarę możliwości w faktyczne bloki do zapisania w chunk store.
- Dzięki temu, że podbloki są mniejsze osiągniemy lepszy współczynnik deduplikacji

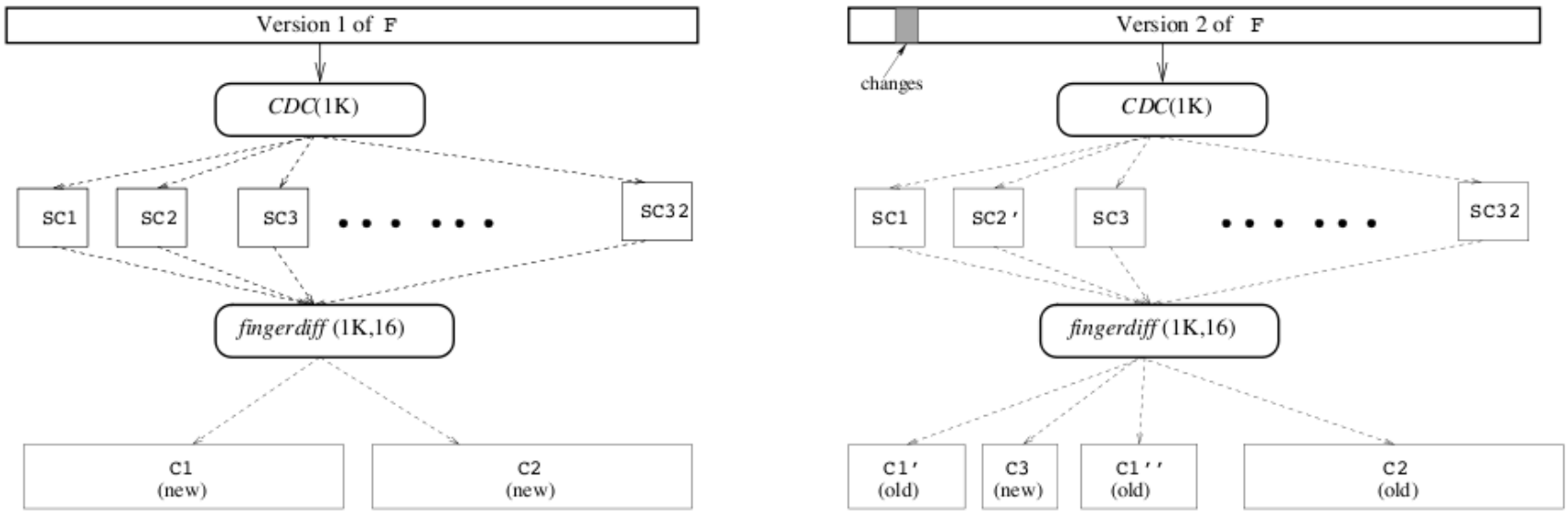
Pomysł

- Wykorzystamy CDC do dzielenia strumienia na mniejsze podbloki, które potem będziemy łączyć w miarę możliwości w faktyczne bloki do zapisania w chunk store.
- Dzięki temu, że podbloki są mniejsze osiągniemy lepszy współczynnik deduplikacji
- Dla każdego pliku czy elementu przechowywanego przez klienta będziemy utrzymywać dodatkowe informacje o jego podblokach, czyli klucz jego nadbloku, offset i długość podbloku (łącznie 12 bajtów metadanych na podblok)

Pomysł

- Wykorzystamy CDC do dzielenia strumienia na mniejsze podbloki, które potem będziemy łączyć w miarę możliwości w faktyczne bloki do zapisania w chunk store.
- Dzięki temu, że podbloki są mniejsze osiągniemy lepszy współczynnik deduplikacji
- Dla każdego pliku czy elementu przechowywanego przez klienta będziemy utrzymywać dodatkowe informacje o jego podblokach, czyli klucz jego nadbloku, offset i długość podbloku (łącznie 12 bajtów metadanych na podblok)
- Stare podbloki, jeśli są spójną częścią dużego bloku który częściowo uległ zmianie, nie trzeba ponownie zapisywać. Trzeba za to zapisać ich offset, długość i hasz nadbloku

Schemat


Version 1 of F
CDC(1K)
SC1
SC2
SC3
SC32
fingerdiff (1K,16)
C1
(new)
C2
(new)
Version 2 of F
changes
CDC(1K)
SC1
SC2'
SC3
SC32
fingerdiff (1K,16)
C1'
(old)
C3
(new)
C1''
(old)
C2
(old)

[3]

Fingerdiff

- Jeśli mamy sekwencję nowych podbloków to łączymy je w jeden blok

Fingerdiff

- Jeśli mamy sekwencję nowych podbloków to łączymy je w jeden blok
- Zamiast maksymalnej długości bloku, będziemy mieć parametr określający maksymalną ilość podbloków

Fingerdiff

- Jeśli mamy sekwencję nowych podbloków to łączymy je w jeden blok
- Zamiast maksymalnej długości bloku, będziemy mieć parametr określający maksymalną ilość podbloków
- Informacje o podblokach przechowujemy jedynie dla pojedynczego obiektu, ale nie warstwa przechowująca fragmenty powinna nadal wychwytywać powtórzenia pomiędzy różnymi obiektami.

Informacje o podblokach

- Pierwsza wersja w tablicy haszującej

Informacje o podblokach

- Pierwsza wersja w tablicy haszującej
- Zorganizowane w drzewo indeksowane kolejnymi bajtami klucza

Informacje o podblokach

- Pierwsza wersja w tablicy haszującej
- Zorganizowane w drzewo indeksowane kolejnymi bajtami klucza
- W liściach są informacje o nadblokach
- Węzły wewnętrzne dzielą się na pełne i rzadkie

Informacje o podblokach

- Pierwsza wersja w tablicy haszującej
- Zorganizowane w drzewo indeksowane kolejnymi bajtami klucza
- W liściach są informacje o nadblokach
- Węzły wewnętrzne dzielą się na pełne i rzadkie
- Jeśli z węzła miałaby prowadzić 1 prosta ścieżka jest ona skompresowana do 1 wierzchołka

Problem

- Te dane muszą być ściągane całe do pamięci RAM
- W tej realizacji nie ma sensu ściągać np. fragmentu drzewa
- Problem ze skalowalnością - rozmiar drzewa będzie rósł wraz z ilością wersji obiektu

Testy

Ilość wykorzystanego miejsca łącznie

	Sources				Databases	Binaries	Total
benchmark	*gcc*	*gdb*	*emacs*	*linux*	*freedb*	*gaim*	
cdc-2k	1414	501	327	1204	396	225	4067
cdc-256	866	363	258	708	348	245	2788
cdc-128	828	344	259	629	369	301	2731
cdc-64	859	358	281	692	442	447	3079
cdc-32	979	500	457	985	644	527	4090
fd-2k	1400	498	324	1195	370	213	3999
fd-256	799	336	239	644	396	196	2611
fd-128	680	293	220	520	317	208	2238
fd-64	579	255	199	469	291	244	2038
fd-32	498	255	221	543	290	246	2052
% saving	40	26	25	23	17	13	25

[3]

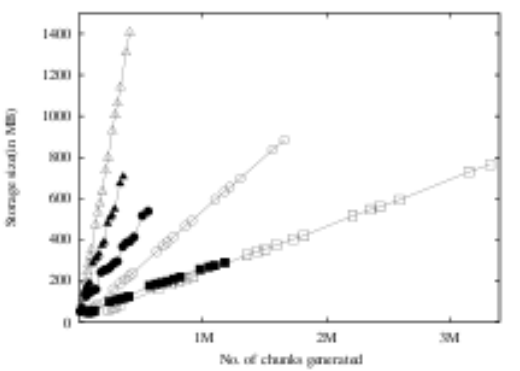
Testy

Ilość wygenerowanych bloków w tyś.

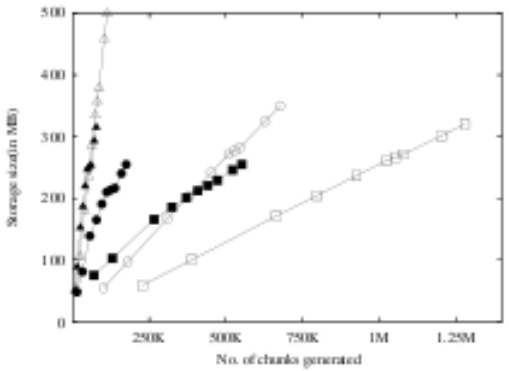
	Sources				Databases	Binaries
benchmark	*gcc*	*gdb*	*emacs*	*linux*	*freedb*	*gaim*
cdc-2k	412	112	84	289	221	193
cdc-256	1880	677	496	1472	1046	1257
cdc-128	3324	1280	971	2467	1799	2310
cdc-64	5799	2325	1864	4579	3616	4589
cdc-32	10204	5231	4804	10202	8226	5760
fd-2k	51	13	10	57	31	22
fd-256	360	76	60	244	419	69
fd-128	642	174	121	515	706	119
fd-64	732	240	203	562	881	231
fd-32	1177	551	398	1059	1309	234

[3]

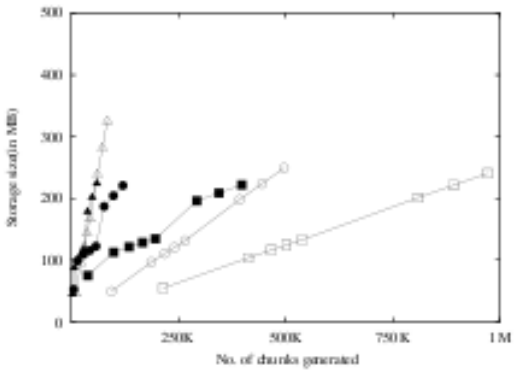
Ilość bloków a wykorzystane miejsce



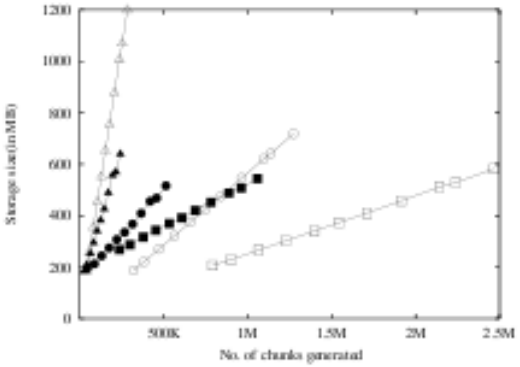
gcc



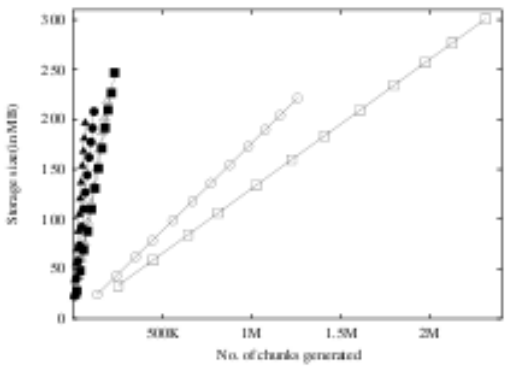
gdb



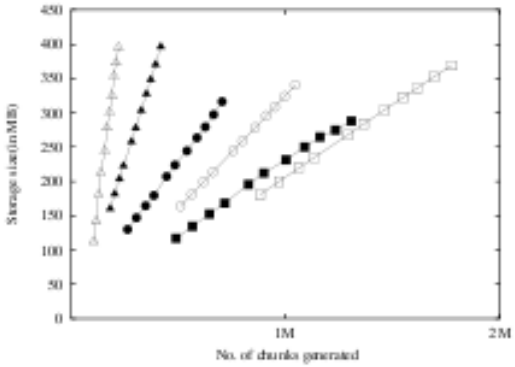
emacs



linux



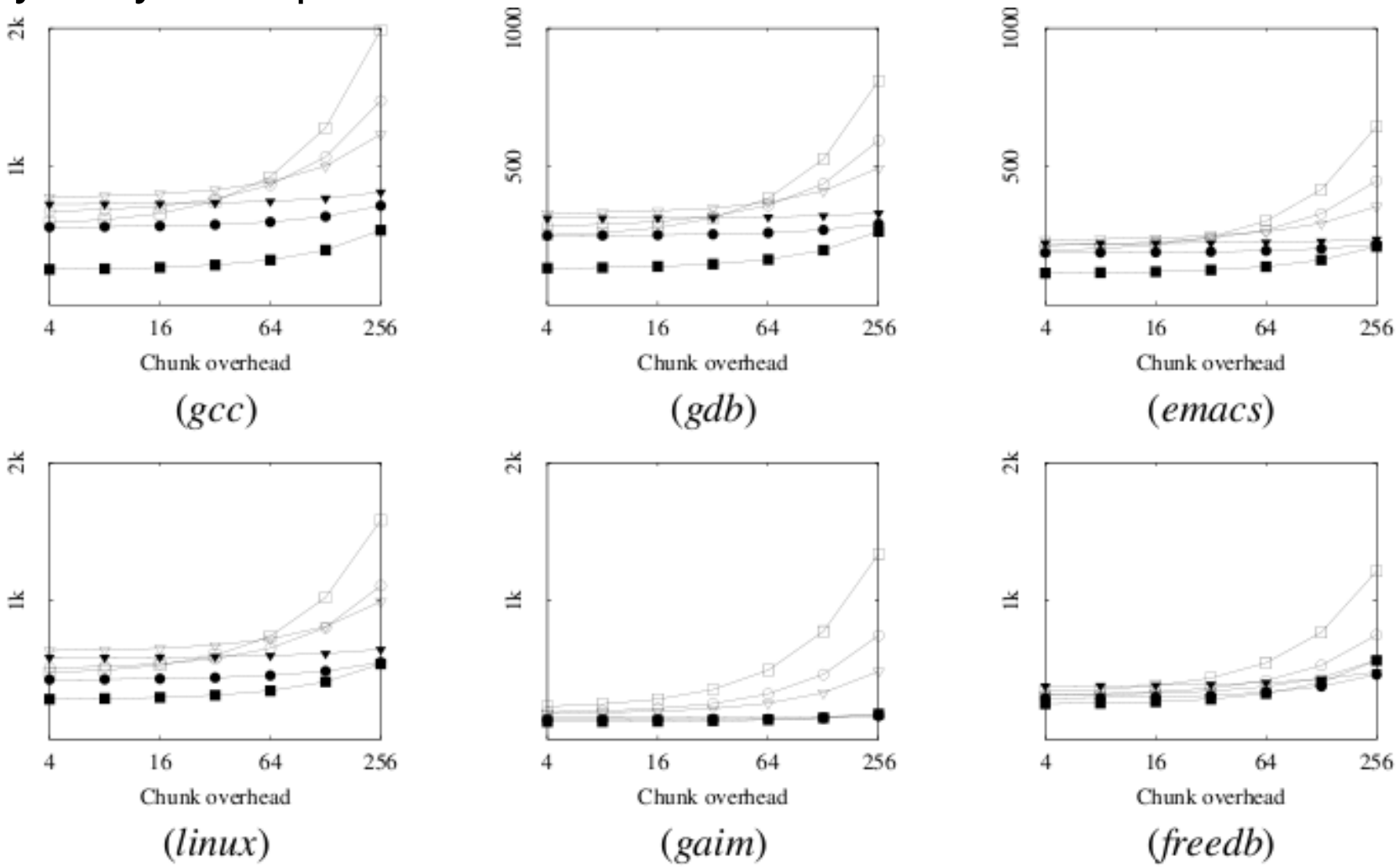
gaim



freedb

cdc-2k
cdc-256
cdc-128
fd-256
fd-128
fd-32

Wykorzystane pasmo sieci



Wpływ ilości bloków na realizację bezpieczeństwa

Zmniejszona ilość ilość bloków jest korzystna, kiedy stosujemy replikację.
Jeśli jednak zastosować kody korygujące błędy, aby zmniejszyć zajętą przestrzeń dyskową trzeba dla każdego bloku przechowywać metadane o każdej jego części.
Wtedy korzyści z redukcji ilości bloków są jeszcze większe.

Pełny indeks

- Kiedy do chunk store przychodzi hasz bloku od klienta należy sprawdzić, czy już go mamy
- Tworzy się więc mapę kluczy, w której mamy zapisane w ich miejsce przechowywania

Pełny indeks

- Kiedy do chunk store przychodzi hasz bloku od klienta należy sprawdzić, czy już go mamy
- Tworzy się więc mapę kluczy, w której mamy zapisane w ich miejsce przechowywania
- Dla praktycznych zastosowań nie mieści się ona w pamięci RAM, trzeba umieścić ją na dysku

Pełny indeks

- Kiedy do chunk store przychodzi hasz bloku od klienta należy sprawdzić, czy już go mamy
- Tworzy się więc mapę kluczy, w której mamy zapisane w ich miejsce przechowywania
- Dla praktycznych zastosowań nie mieści się ona w pamięci RAM, trzeba umieścić ją na dysku
- Dla każdego bloku musimy wykonać operację seek w indeksie
- Jest to problem wąskiego gardła

Pełny indeks

- Kiedy do chunk store przychodzi hasz bloku od klienta należy sprawdzić, czy już go mamy
- Tworzy się więc mapę kluczy, w której mamy zapisane w ich miejsce przechowywania
- Dla praktycznych zastosowań nie mieści się ona w pamięci RAM, trzeba umieścić ją na dysku
- Dla każdego bloku musimy wykonać operację seek w indeksie
- Jest to problem wąskiego gardła
- Kluczowa jest tu własność lokalności

Lokalność

- Nie ma tu lokalności w typowym sensie (po jednym A nie nastąpi kolejne)

Lokalność

- Nie ma tu lokalności w typowym sensie (po jednym A nie nastąpi kolejne)
- Jeśli po A występuje B i C to po kolejnym A też będą B i C
- Nawet jeśli to drugie A jest 10GB dalej

Proponowane rozwiązanie

- Filtr Blooma
- Układ segmentów zgodny z budową strumienia
- Cache zgodny z zasadą lokalności

Pozwalają one łącznie zredukować liczbę odczytów z dysku o nawet 99%.

Filtr Blooma

- Jak działa filtr Blooma już wiadomo

Filtr Blooma

- Jak działa filtr Blooma już wiadomo
- Wykorzystamy go aby nie szukać w indeksie kluczy, których tam na pewno nie ma
- Autorzy proponują wykorzystanie 8 bitów na 1 blok oraz 4 lub 5 funkcji haszujących

Filtr Blooma

- Jak działa filtr Blooma już wiadomo
- Wykorzystamy go aby nie szukać w indeksie kluczy, których tam na pewno nie ma
- Autorzy proponują wykorzystanie 8 bitów na 1 blok oraz 4 lub 5 funkcji haszujących
- Z Filtra Blooma zupełnie nie da się już nic usunąć
- W takiej konfiguracji zużywa dużo pamięci RAM

Układ segmentów

- Jeśli zapisujemy dwa strumienie jednocześnie robimy to osobno do osobnych kontenerów
- Osobno również zapisujemy metadane i samą zawartość danego strumienia
- Dzięki temu sam strumień łatwiej potem odczytać
- Jednym odczytem z dysku wczytujemy metadane o bardzo wielu blokach

Cache indeksu

- Kiedy znajdziemy stary blok w indeksie na dysku wczytujemy cały kontener, który zawiera metadane tego bloku do pamięci RAM
- Cache zorganizowany jest jako tablica haszująca
- Kiedy mamy wykorzystane zbyt dużo pamięci usuwamy wszystkie wpisy z nieefektywnego kontenera
- Decyduje tu polityka LRU

Podsumowując

Kiedy mamy sprawdzić blok:

- sprawdź w cache
- sprawdź filtr blooma
- sprawdź w indeksie, uaktualnij cache indeksu
- jeśli nie znaleziono, blok jest nowy, zapisz go

Testy

	Exchange data		Engineering data	
	# disk I/Os	% of total	# disk I/Os	% of total
no Summary Vector and no Locality Preserved Caching	328,613,503	100.00%	318,236,712	100.00%
Summary Vector only	274,364,788	83.49%	259,135,171	81.43%
Locality Preserved Caching only	57,725,844	17.57%	60,358,875	18.97%
Summary Vector and Locality Preserved Caching	3,477,129	1.06%	1,257,316	0.40%

[4]

Podsumowując

- Filtr Blooma jest efektywny kiedy zapisujemy pierwszą wersję obiektu
- Cache jest efektywny kiedy zapisujemy kolejne wersje obiektu

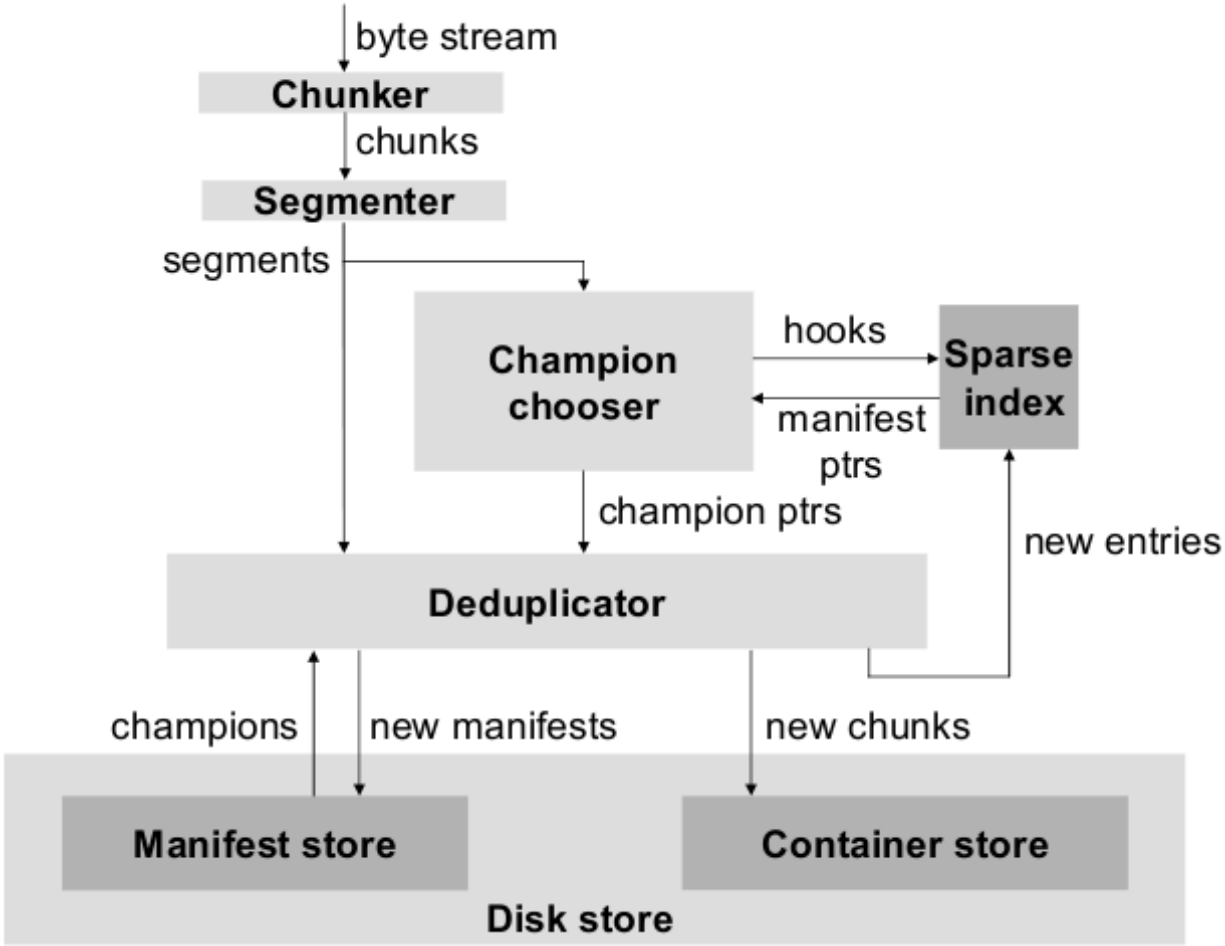
Inne podejście

- Będziemy dalej dążyć do minimalizacji wykorzystania RAM'u
- Nie będziemy już mieć pełnego indeksu, będziemy mieć tylko rzadki indeks w pamięci
- Poświęcimy poprawność wykrywania duplikatów

Inne podejście

- Będziemy dalej dążyć do minimalizacji wykorzystania RAM'u
- Nie będziemy już mieć pełnego indeksu, będziemy mieć tylko rzadki indeks w pamięci
- Poświęcimy poprawność wykrywania duplikatów
- ale dzięki temu, wykorzystamy mniejsze bloki co w sumie powinno poprawić współczynnik deduplikacji

Schemat działania



[5]

Szczegóły

- Do podziału na bloki wykorzystano TTTD
- Jako haczyki służą bloki o haszu którego odpowiednio dużo bitów jest zerowych

Szczegóły

- Do podziału na bloki wykorzystano TTTD
- Jako haczyki służą bloki o haszu którego odpowiednio dużo bitów jest zerowych
- Podział na segmenty realizowany był albo wg ustalonej ilości bloków
- lub aby unikać problemy z przesuwaniem granic segmentacji o zmiennej długości
- Bloków nie dzieli się na kawałki

Szczegóły

- Do podziału na bloki wykorzystano TTTD
- Jako haczyki służą bloki o haszu którego odpowiednio dużo bitów jest zerowych
- Podział na segmenty realizowany był albo wg ustalonej ilości bloków
- lub aby unikać problemy z przesuwaniem granic segmentacji o zmiennej długości
- Bloków nie dzieli się na kawałki
- Zamiast odcisku palca okna, wykorzystuje się hasz bloku
- Wykorzystuje się tu TTTD, tyle że na poziomie bloków

Wybór zwycięzców

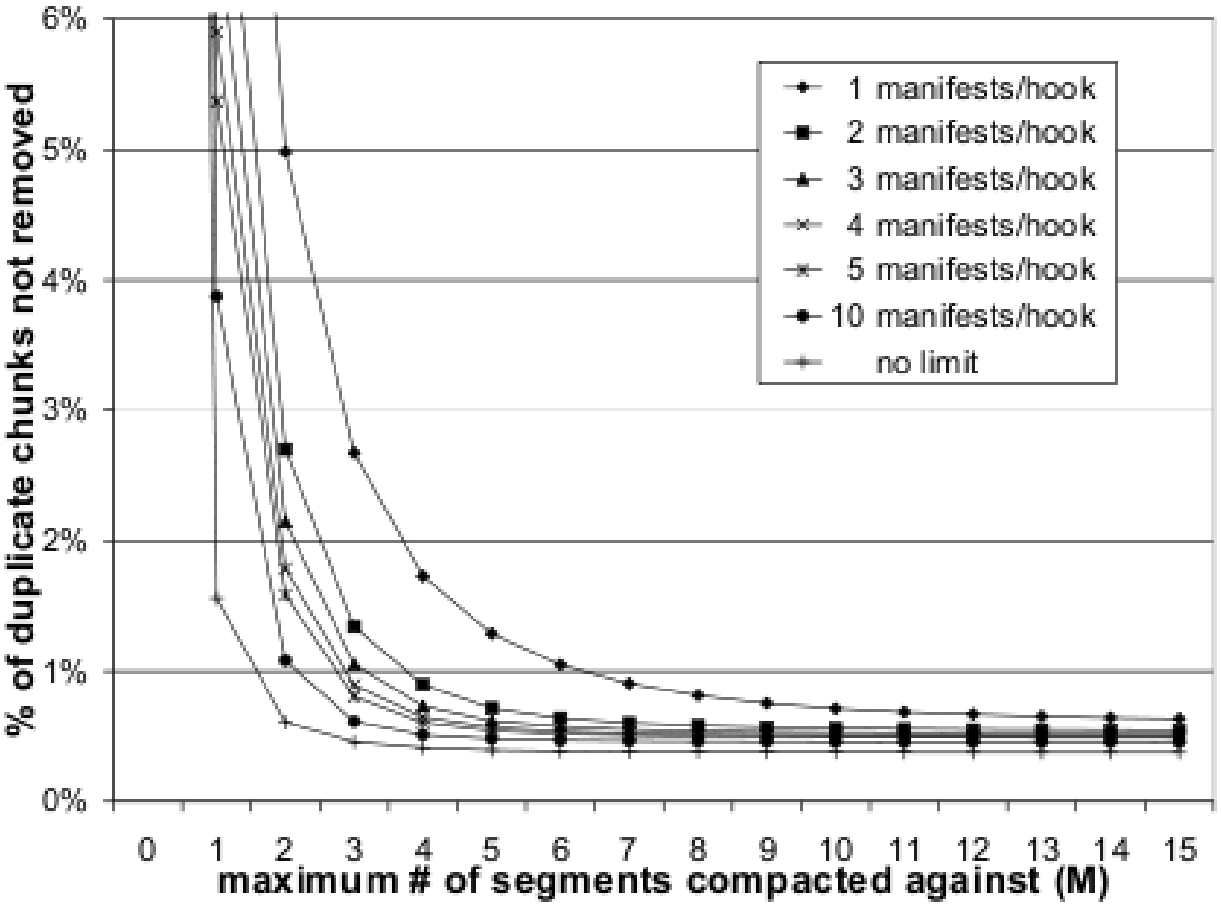
- Po znalezieniu w rzadkim indeksie manifestów zawierających haczyki, wybierzemy tylko kilka najlepszych z nich
- Wybieramy kolejno przyznając punkty za każde wystąpienie jeszcze nie pokrytego haczyka
- W przypadku remisu wygrywa nowszy manifest

Rzadki indeks

- Po zapisaniu nowego manifestu dodajemy go do indeksu
- Ograniczamy ilość manifestów dla jednego haczyka, usuwając najstarsze manifesty

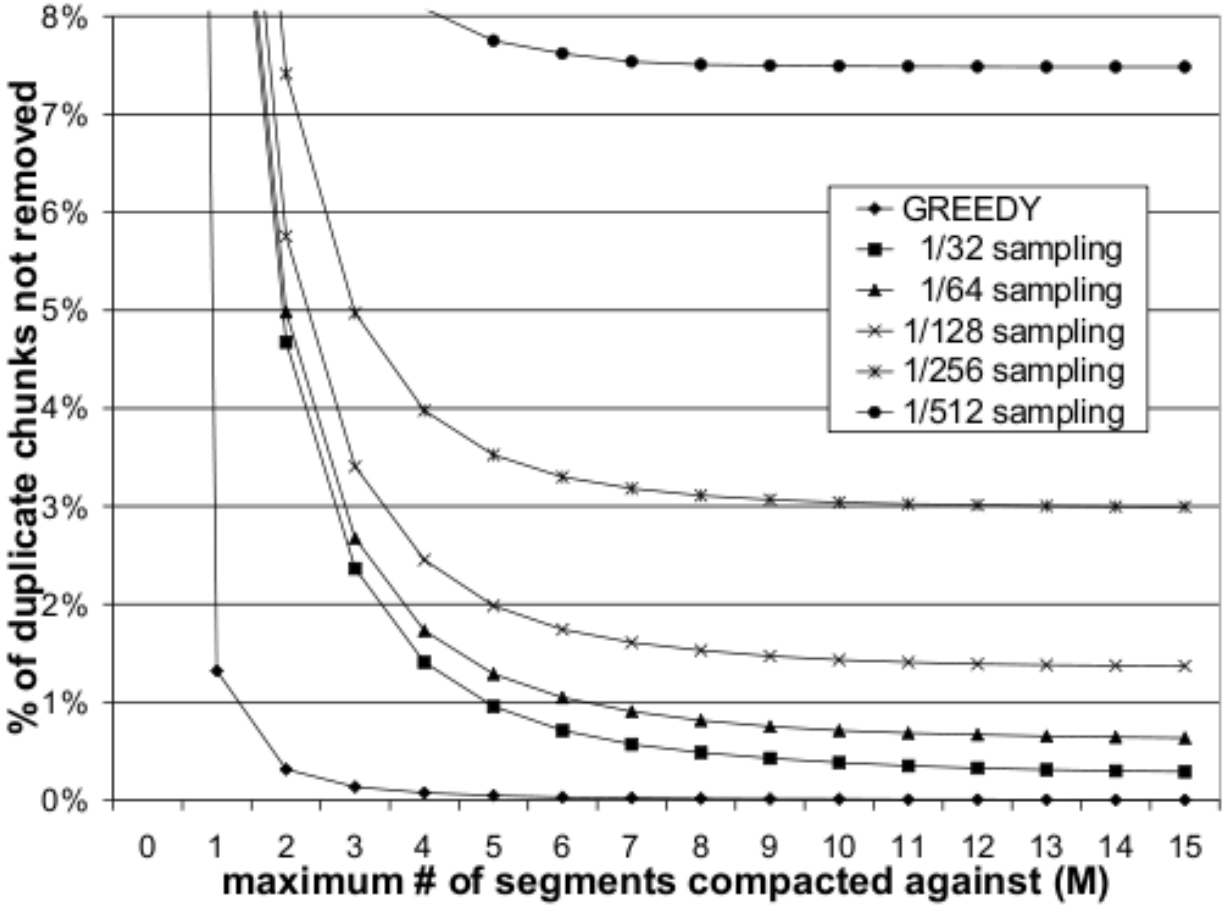
W testach rozmiar bloku to zazwyczaj 4kB. Rozmiar segmentu 10MB. Ilość zwycięzców 10.

Testy


% of duplicate chunks not removed
6%
5%
4%
3%
2%
1%
0%
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
maximum # of segments compacted against (M)
1 manifests/hook
2 manifests/hook
3 manifests/hook
4 manifests/hook
5 manifests/hook
10 manifests/hook
no limit

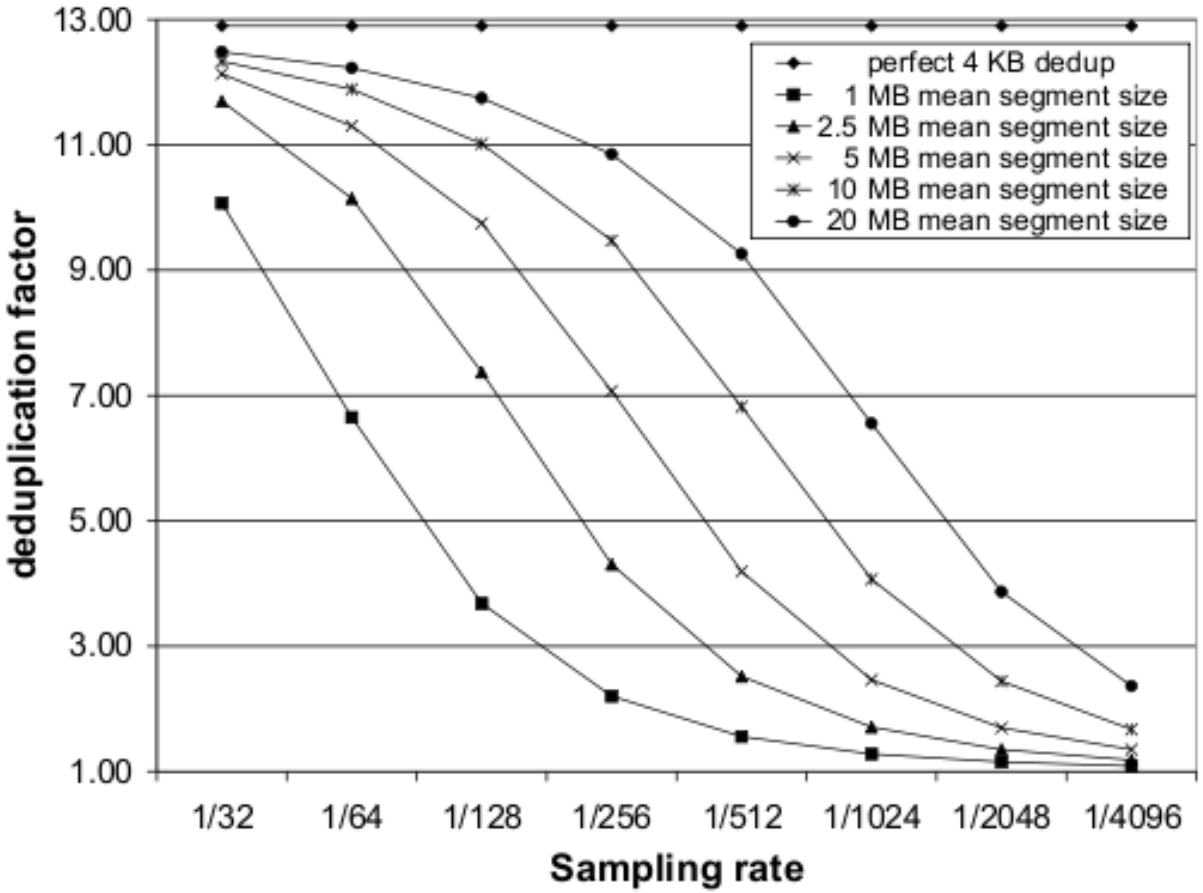
[5]

Testy


% of duplicate chunks not removed
8%
7%
6%
5%
4%
3%
2%
1%
0%
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
maximum # of segments compacted against (M)
GREEDY
1/32 sampling
1/64 sampling
1/128 sampling
1/256 sampling
1/512 sampling

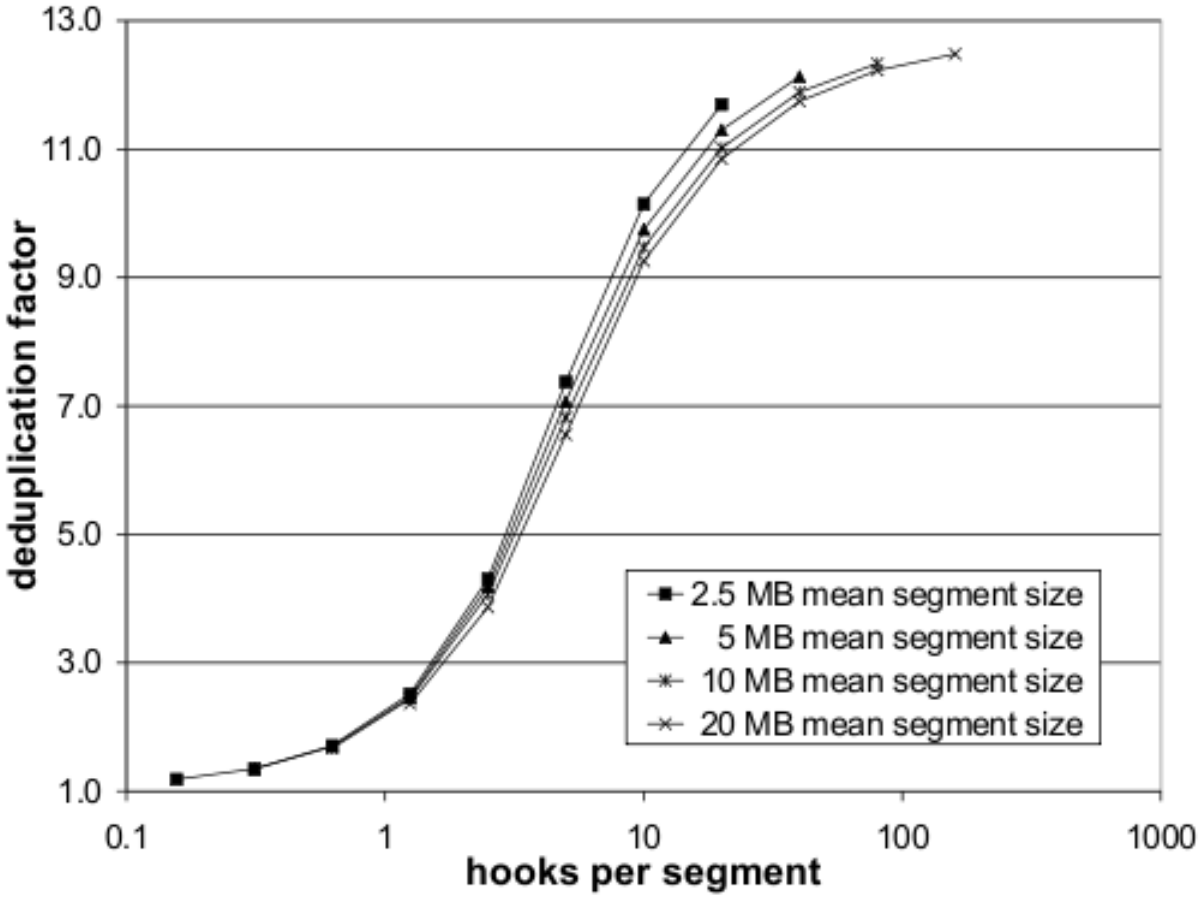
[5]

Testy


13.00
11.00
9.00
7.00
5.00
3.00
1.00
deduplication factor
perfect 4 KB dedup
1 MB mean segment size
2.5 MB mean segment size
5 MB mean segment size
10 MB mean segment size
20 MB mean segment size
1/32
1/64
1/128
1/256
1/512
1/1024
1/2048
1/4096
Sampling rate

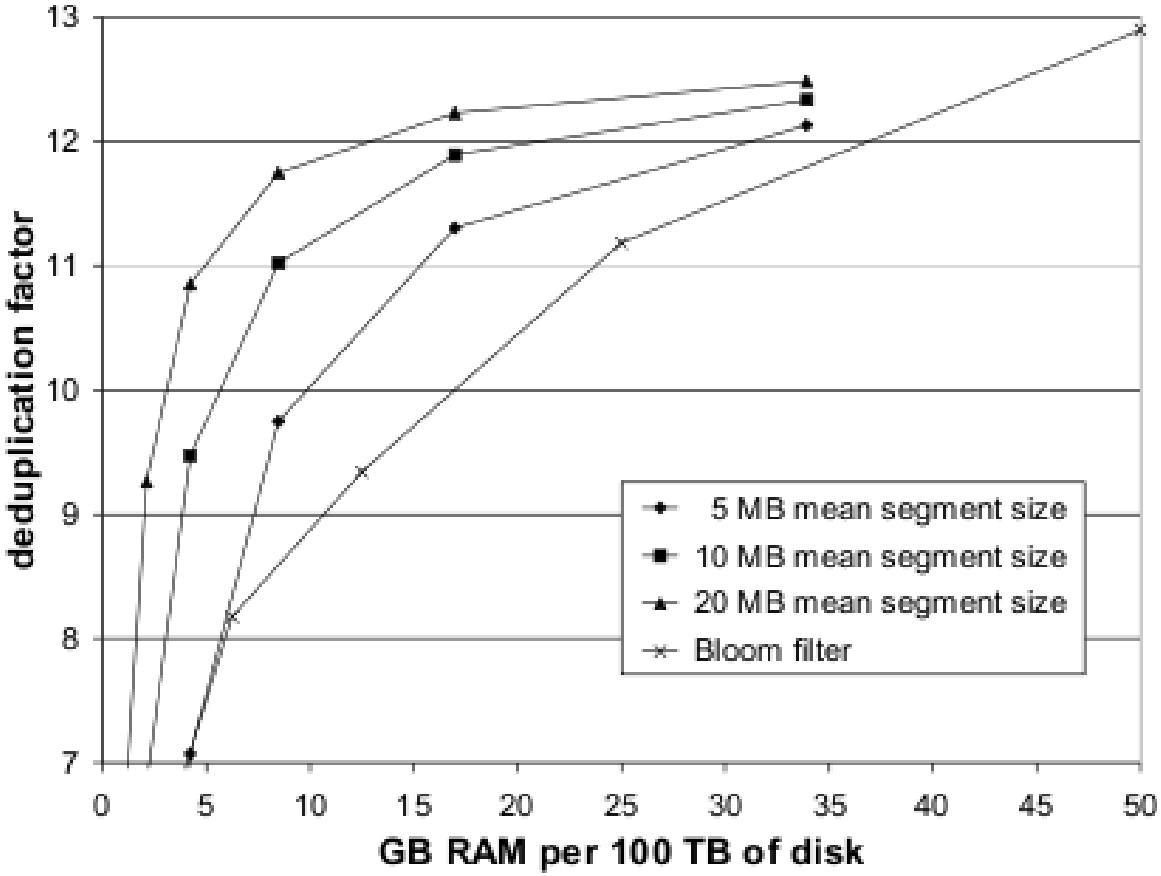
[5]

Testy


13.0
11.0
9.0
7.0
5.0
3.0
1.0
deduplication factor
0.1
1
10
100
1000
hooks per segment
2.5 MB mean segment size
5 MB mean segment size
10 MB mean segment size
20 MB mean segment size

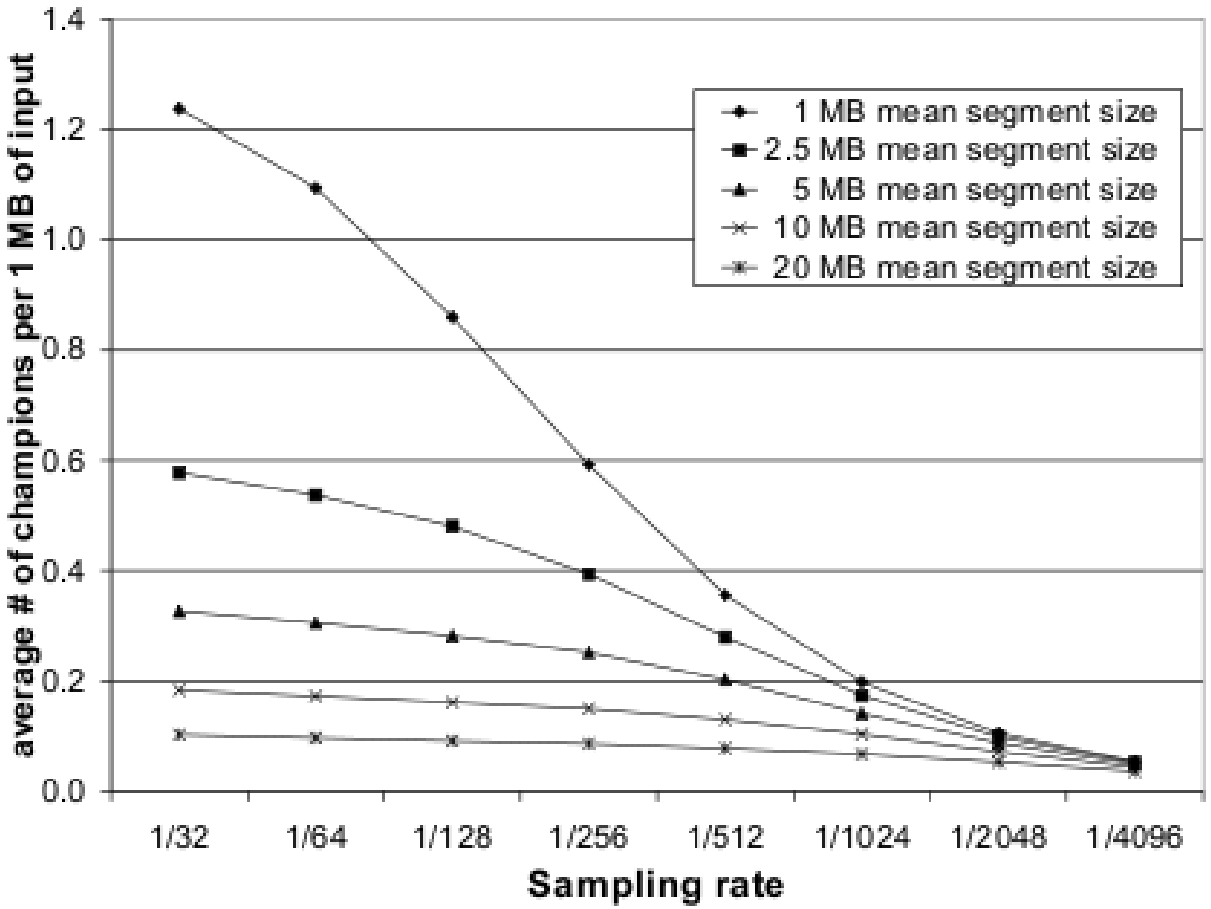
[5]

Porównanie obu metod



[5]

Testy


1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
average # of champions per 1 MB of input
1 MB mean segment size
2.5 MB mean segment size
5 MB mean segment size
10 MB mean segment size
20 MB mean segment size
1/32
1/64
1/128
1/256
1/512
1/1024
1/2048
1/4096
Sampling rate

[5]

Alternatywy

Oprócz wykrywania fragmentów, które są identyczne można próbować znajdować podobne elementy i zapisywać samą różnicę między nimi (tzw. delta encoding), tak działają systemy kontroli wersji.

Zastosowania

- Głównie systemy do archiwizacji
- LBFS
- ogólnie w miejscach gdzie chcemy unikać przesyłania dużych porcji danych przez sieć, które ktoś już może mieć

Koniec

Dziękuję za uwagę.

Źródła

1. “A Framework for Analyzing and Improving Content-Based Chunking Algorithms” Kave Eshghi, Hsiu Khuern Tang
2. “Rabin’s Fingerprinting” Tong Zhous http //discovery.csc.ncsu.edu/ãliu3/reading_group/Rabin%27s%2
3. “Improving Duplicate Elimination in Storage Systems“ Deepak R. Bobbarjung, Suresh Jagannathan and Cezary Dubnicki
4. “Avoiding the Disk Bottleneck in the Data Domain Deduplication File System“ Benjamin Zhu, Kai Li and Hugo Patterson
5. “Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality“ Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise and Peter Camble