Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Michał Niciejewski
Nr albumu: 394618

Krzysztof Małysa
Nr albumu: 394442

Aliaksandr Sarokin
Nr albumu: 372525

Wojciech Mitros
Nr albumu: 394488

Asynchroniczny system plików w przestrzeni użytkownika zoptymalizowany pod duże pliki

Praca licencjacka
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra Jacka Sroki
Instytut Informatyki

Warszawa, Czerwiec 2020

Streszczenie

Niniejsza praca przedstawia prototypową implementację asynchronicznego systemu plików dla bazy danych Scylla od ScyllaDB. System jest napisany w przestrzeni użytkownika. Charakteryzuje się tym, że operacje na dysku są dopisywane sekwencyjnie na końcu struktury przypominającej log, co zwiększa efektywność przy obsłudze dużych zapisów. W celu osiągnięcia dobrej skalowalności i zgodnie z ideami wątek na rdzeń i brak współdzielenia zasobów (*ang.* shared-nothing), każdy wątek obsługuje operacje na pewnej, nieprzecinającej się z innymi wątkami, części drzewa katalogu redukując przez to synchronizację.

Nasz system plików porównaliśmy z innymi znanymi systemami – XFS oraz ext4. W testach symulujących działanie systemu plików w dłuższym przedziale czasu różnica czasu wykonania testu w porównaniu do innych systemów wynosi poniżej 5% w przypadkach przypominających działanie Scylli i poniżej 10% w przypadku ogólnym. Przy testowaniu opóźnienia (*ang.* latency) operacji, nasz system w większości przypadków osiąga wyniki nawet do 50% lepsze w porównaniu z XFS i 45% lepsze od ext4.

Słowa kluczowe

system plików, przestrzeń użytkownika, asynchroniczność, opóźnienie, struktura logu, podział na rdzenie

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

10011007.10010940.10010941.10010949.10003512 — Software and its engineering – Software organization and properties – Contextual software domains – Operating systems – File systems management

Tytuł pracy w języku angielskim

Asynchronous user space file system optimised for big files

Spis treści

Wprowadzenie 5

1. Wybrana platforma programistyczna 9

2. Architektura systemu plików 11
2.1. System plików o strukturze logu 11
2.2. Synchronizacja pomiędzy operacjami asynchronicznymi 12
2.3. Realizacja odczytów i zapisów 13
2.4. Podział na rdzenie 15
2.4.1. Podział dysku 15
2.4.2. Operacje na instancji 15
2.4.3. Współdzielony katalog główny 15
2.4.4. Operacje na pliku 16
2.5. Kompakcja 16
2.5.1. Wybór klastrów do kompakcji 17
2.5.2. Realizacja kompakcji 18

3. Testy efektywności i analiza wyników 19
3.1. Porównanie parametrów SeastarFS 19
3.1.1. Wielkość klastra 20
3.1.2. Strategie kompakcji 21
3.2. Porównanie z XFS i ext4 21
3.2.1. Porównanie w symulacji 21
3.2.2. Porównanie opóźnienia 23

4. Podsumowanie i kierunki rozwoju 27
4.1. Kierunki rozwoju 27
4.1.1. Defragmentacja danych w pliku 27
4.1.2. Kompakcja logu metadanych 27
4.1.3. Dopasowanie podziału zasobów przy zmianie liczby rdzeni 28
4.2. Podziękowania 28

A. Organizacja pracy zespołu 29
A.1. Podział obowiązków 29
A.2. Metody organizacji pracy 30
A.3. Współpraca z klientem 30

B. Zawartość płyty CD 31

C. Testy poprawnościowe i atrapy 33
C.1. Testy jednostkowe 33
C.2. Atrapy 33

Bibliografia 35

Wprowadzenie

Jednym ze sposobów optymalizacji baz danych jest dostosowanie ich do możliwości nowoczesnego sprzętu. W ciągu ostatnich lat nastąpiła zmiana charakterystyki dysków twardych [11]. Wcześniej dyski HDD miały elementy mechaniczne tj. głowicę, przez co losowa zmiana miejsca odczytu / zapisu na dysku trwała od kilku do kilkudziesięciu milisekund. Prędkość przesyłu danych (*ang.* throughput) nawet nowszych dysków HDD wynosi jedynie 100-150 MB/s. W dzisiejszych dyskach SSD koszt zmiany miejsca odczytu jest rzędu 0.1 milisekundy, więc jest w zasadzie pomijalny. Prędkość przesyłu danych wzrosła do ponad 2 GB/s. Wymagało to również zmiany protokołu komunikacji z dyskiem z AHCI na NVMe [20]. Dodatkowo częstotliwość taktowania procesorów niemalże przestała rosnąć [21] (teraz są to okolice 4 GHz i rocznie wzrasta o ok. 0.1-0.3 Ghz). Zamiast tego liczba rdzeni stale rośnie (obecne już są procesory z 64 rdzeniami [1]) [4]. Efektywne oprogramowanie musi brać pod uwagę dyski SSD i wielordzeniowe procesory.

Oprócz tego, bazy danych muszą dostosowywać się do zmieniającego się oprogramowania którego używają, w tym systemów operacyjnych. Dla przykładu dodane w wersji Linux 5.1 (maj 2019) io_uring [2, 3] pozwala na zmniejszenie liczby zapytań systemowych przy operacjach wejścia-wyjścia.

Nie wszystkie rozwiązania realizowane w systemach operacyjnych są dobre dla baz danych. Przeniesienie pewnych zadań systemu operacyjnego do bazy danych czasem daje mierzalne korzyści, np. realizacja w niej pamięci podręcznej pozwala osiągnąć większy współczynnik trafień. Jest tak, gdyż baza danych ma więcej kontekstu niż system operacyjny.

Cel projektu

Scylla jest bazą danych NoSQL zapewniającą bardzo niskie opóźnienie operacji, która jest alternatywą dla systemu Apache Cassandra, ale z nawet 10-krotnie większą przepustowością (*ang.* throughput) [16, 13, 19].

Projekt miał charakter badawczy mający na celu stworzenie asynchronicznego systemu plików w przestrzeni użytkownika pozwalającego na efektywną eksploatację nowoczesnych dysków SSD w przypadkach użycia występujących w bazie danych Scylla oraz zbadanie jak wypada na tle uznanych systemów plików takich jak ext4 i XFS. Więcej na temat wspomnianych przypadków użycia można znaleźć w podrozdziale 2.3 i [14].

System ma być używany jedynie w wąskim zakresie zastosowań, więc nie musi być zgodny ze standardem POSIX.

Ze względu na specyficzny sposób funkcjonowania Scylli, pewne operacje powinny być obsługiwane przez system plików szczególnie sprawnie, nawet jeśli skutkuje to wolniejszym działaniem w pozostałych przypadkach, na przykład dopisywanie dużych i małych fragmentów danych na koniec pliku powinno być wykonywane z minimalnym możliwym opóźnieniem, a zapisy nie na koniec pliku mogą mieć wielokrotnie większy narzut bez nadmiernej szkody dla

szybkości bazy danych.

Motywacja

Firma ScyllaDB zlecając projekt uznała, że system plików pod Scyllę powinien spełniać dwie cechy – być asynchroniczny i napisany w przestrzeni użytkownika. W kolejnych sekcjach przedstawione jest uzasadnienie tej decyzji.

Asynchroniczność

Aby zyskać możliwie najwięcej ze zdolności równoległego wykonywania operacji przez dysk, system plików musi obsługiwać wiele odczytów/zapisów równocześnie. Standardową metodą rozwiązania tego problemu jest tworzenie nowych wątków procesu, w których zlecone operacje wejścia-wyjścia wykonywane są synchronicznie. Wiąże się to z koniecznością wykonywania wielu przełączeń kontekstu między nimi. Alternatywnym podejściem eliminującym tę potrzebę jest samodzielny podział wątku programu na niezależne fragmenty, które są przerywane w momencie wysłania zapytania do systemu plików i wznawiane, kiedy odpowiedź zostanie wygenerowana. W czasie wykonywania operacji na systemie plików procesor może wykonywać inne, niezależne części programu. To pozwala na działanie wielu operacji wejścia-wyjścia w jednym momencie bez konieczności tworzenia nowych wątków.

Systemy plików używane przez Scyllę muszą dobrze obsługiwać asynchroniczne zapytania. Brak obsługi asynchroniczności powoduje niepożądane wzrosty opóźnienia operacji.

Na dzień 2020-05-30 żaden z systemów plików XFS, ext4, Btrfs, VFAT, exFAT, F2FS, JFS, NILFS2, NTFS, ReiserFS, UDF nie jest asynchroniczny w pełni [10]. To pokazuje, że nasz system faktycznie wypełnia pewną lukę i ma szanse osiągnąć lepsze opóźnienia w stosunku do wspomnianych systemów.

Implementacja w przestrzeni użytkownika

Standardowe systemy plików jak ext4, czy XFS działają w przestrzeni jądra. Implementacja w przestrzeni użytkownika ma kilka znaczących zalet:

- Jest to istotnie prostsze niż implementacja w przestrzeni jądra. W przestrzeni użytkownika, znacznie łatwiej programować asynchronicznie. Pisanie złożonego, czysto asynchronicznego kodu w C jest uciążliwe, a to jest jedyna opcja gdy modyfikujemy jądro. Dodatkowo, należałoby nasz system plików zintegrować w jądro, co w wielu miejscach jest co najmniej nieoczywiste, jeżeli nie bardzo trudne. Dodatkową kwestią, która by nas ograniczała w przestrzeni jądra jest to, że nasz system plików musiałby się dostosować do POSIX, a wybraliśmy że tego nie chcemy robić, gdyż wymagałoby to dużo większego nakładu pracy.

- Lepsza przenośność – Nie potrzeba modyfikować wersja jądra aby korzystać z systemu plików zaimplementowanego w przestrzeni użytkownika. Dodatkowo rozwój systemu plików nie jest uzależniony od rozwoju jądra.

- Mniejsza liczba wywołań systemowych (*ang.* syscall) – Przy odpowiedniej implementacji operowanie na metadanych wymaga tylko sporadycznych wywołań systemowych. W przypadku jądra każda taka operacja musi być przekazana do jądra.

  Operacje czytania i zapisu mogą zostać rozbite na kilka mniejszych operacji które zostaną przekazane jądru. Gdyby system plików był częścią jądra, to to rozbicie odbyłoby

się w jądrze, i tylko jedna operacja przeszłaby przez barierę użytkownik-jądro, a nie kilka; niemniej zapytania które dostaje dysk będą te same. Jednakże, tę wadę bardzo mocno niweluje fakt, że istnieją mechanizmy takie jak io_uring [2, 3], które pozwalają przesyłać wiele żądań wejścia-wyjścia z użyciem kilka rzędów wielkości mniejszej liczby zapytań systemowych niż operacji.

- Czas obrotu operacji – Systemy plików zaimplementowane w przestrzeni jądra czasami generują wiele żądań wejścia-wyjścia do dysku i nie da się tego kontrolować z przestrzeni użytkownika. Zaimplementowanie systemu plików w przestrzeni użytkownika pozwala szeregować wedle własnego uznania również zapytania wejścia-wyjścia generowane przez system plików, co jest bardzo istotne dla bazy danych Scylla [6] – ważne operacje mogą być wykonywane szybko, nawet przy dużym obciążeniu ze strony innych, mniej ważnych operacji [5].

Zawartość pracy

W rozdziale 1 opisano zastosowane w implementacji koncepcje i technologie. W rozdziale 2 przedstawiono opis architektury systemu plików wraz z charakterystyką jego poszczególnych elementów. W rozdziale 3 zaprezentowano wyniki testów efektywności. Podsumowanie projektu, wyrażenie podziękowań w jego prowadzeniu i powstaniu oraz przemyślenia dotyczące dalszego rozwoju zawierają się w rozdziale 4. W dodatku A umieszczono opis organizacji i podziału pracy. W dodatku B wyszczególniono zawartość dołączonej płyty CD. W dodatku C opisano testy poprawnościowe i atrapy stworzone na ich potrzeby.

Rozdział 1

Wybrana platforma programistyczna

Naturalną decyzją co do platformy, na której tworzony będzie projekt, był wybór zrębu Seastar. Pozwala on na pisanie kodu o wszystkich właściwościach wymienionych we Wprowadzeniu. Seastar umożliwia również implementowanie programów, które wykonają się równocześnie na każdym rdzeniu procesora w osobnej instancji tzn. dokładnie jeden wątek na rdzeniu. Taki podział jest optymalny dla naszego systemu – mniejsza liczba wątków nie eksploatuje potencjału procesora, a z powodu sposobu wykonywania asynchronicznych programów w Seastarze, większa ich liczba nie przynosi korzyści. Podejście to wymaga jednak odpowiedniej realizacji [17] – została ona opisana w rozdziale 2.4.

W zrębie Seastar, każdy program napisany asynchronicznie składa się ze spójnych kawałków zwanych kontynuacjami, z których każda wykonuje się bez oddawania procesora. Miejsca, w których procesor jest dobrowolnie oddawany lub gdy trzeba na coś poczekać, np. operację wejścia-wyjścia, są punktami rozdzielającymi kontynuacje. Takie podejście jest bardzo podobne do modelu obietnic (*ang* future-promise) z języka JavaScript [9]. Łańcuchy składające się z wielu kontynuacji tworzą tzw. włókna (ang. *fibers*).

W jednym wątku może być równocześnie przetwarzanych wiele włókien. Ale tylko jedna kontynuacja może działać w danym momencie. Zwykle jest tak, że wiele kontynuacji jest gotowych do wykonania, gdy obecnie wykonywana się skończy. Wtedy Seastar zleca jądru wykonanie operacji, którą zakończyła się kontynuacja (jeżeli jądro jest do tego potrzebne) oraz wybiera do wykonania następną gotową kontynuację, czyli dokonuje szeregowania zadań. Zastosowane szeregowanie pozwala w pełni spożytkować możliwości dysków SSD, które stają się nasycone dopiero przy kilkuset operacjach wykonywanych jednocześnie [7].

Seastar jest platformą, na której powstała baza danych Scylla – docelowy użytkownik systemu. Implementacja systemu plików w tym zrębie znacznie ułatwi jego integrację, na czym zależy zlecającemu.

Rozdział 2

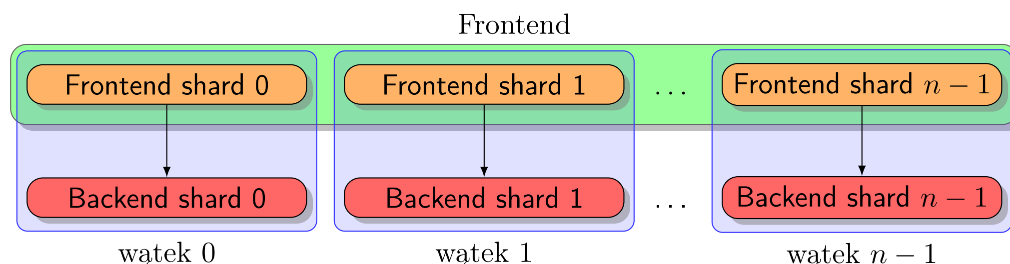
Architektura systemu plików

W rozdziale przedstawimy architekturę stworzonego przez nas systemu plików – SeastarFS.

System plików jest podzielony na instancje. Każda instancja działa na jednym wątku i składa się z instancji frontendu i instancji backendu. Instancje backendu służą do niskopoziomowej realizacji logiki operacji systemu plików i wykonują operacje dyskowe.

Użytkownik chcąc wykonać operację, zleca ją pojedynczej instancji frontendu. Z instancją backendu, komunikuje się jedynie odpowiadająca mu instancja frontendu. Instancje frontendu potrafią się ze sobą komunikować, dokładniej jest to opisane w rozdziale 2.4.

Rysunek 2.1 przedstawia opisany schemat.



Rysunek 2.1: Schemat podziału architektury systemu plików na wątki oraz instancje frontendu i backendu. Instancje frontendu potrafią się ze sobą komunikować, co zostało pokazane zgrupowaniem ich w jeden byt: Frontend.

2.1. System plików o strukturze logu

Log jest to ciąg bajtów, którego jedyną modyfikacją jest dopisanie danych na końcu, dane które zostały wcześniej zapisane nie są potem modyfikowane. Systemy plików o strukturze logu składają się z jednego lub kilku logów [12]. SeastarFS składa się z dwóch logów: logu metadanych i logu danych.

Każda operacja modyfikacji systemu plików będzie reprezentowana jako rekord w logu metadanych. Przykładowo utworzenie czy usunięcie pliku to dopisanie na koniec logu metadanych rekordu opisującego tę operację. W typowych systemach plików takich jak ext4, taki log odpowiada mniej więcej dziennikowi, ale tam pełni on jedynie funkcję zabezpieczenia przed awariami, np. niespodziewaną utratą zasilania. Główna różnica polega na tym, że w SeastarFS metadane są zapisywane jedynie w logu.

Do logu danych trafia jedynie zawartość zapisów, przy czym nie każdy zapis trafia do logu danych – dokładniejszy opis znajduje się w podrozdziale 2.3.

Z powodu struktury logu, rekordy opisujące metadane dotyczące jednego pliku mogą być rozrzucone po całym dysku. Aby uniknąć kosztownego odczytywania wielu małych fragmentów z wielu miejsc w celu odczytania metadanych, wszystkie metadane są trzymane w pamięci operacyjnej. Zakładamy przy tym, że tych metadanych nie będzie za dużo, co jest uzasadnione tym, że nasz system plików jest tworzony pod Scyllę, która nie operuje na dużych ilościach metadanych.

Ponieważ liczba logów jest większa niż jeden i dyski nie mają nieskończonej pojemności to realizacja logów w przestrzeni dyskowej nie jest oczywista. Sposobem wybranym przez nas jest podzielenie przestrzeni dyskowej na kawałki równej wielkości – tak zwane klastry. W ramach jednego klastra log działa normalnie, ale gdy klaster się kończy, to log jest kontynuowany w innym klastrze. Informacja o tym, gdzie log metadanych jest kontynuowany jest zapisywana jako specjalny rekord do logu metadanych. Dla logu danych taka informacja nie jest potrzebna.

Wystartowanie systemu plików od zera tzw. ”bootstrapping” polega na odczytaniu całego logu metadanych i odtworzeniu operacji, które zostały w nim zapisane.

2.2. Synchronizacja pomiędzy operacjami asynchronicznymi

Jedna instancja backendu (backend shard) działa w jednym wątku, więc nie ma problemów ze współbieżnością w stosunku do innych wątków. Jednak z powodu asynchroniczności niemal każdej operacji zachodzi potrzeba ochrony / synchronizacji pomiędzy takimi operacjami w ramach jednego wątku.

Jedną z operacji wymagającej synchronizacji jest tworzenie pliku, które potrzebuje dwóch asynchronicznych operacji: stworzenia nowego i-węzła (*ang.* i-node) oraz dodania wpisu katalogowego. Bez ochrony, wiele takich operacji uruchomionych jednocześnie mogłoby spowodować uszkodzenie struktury metadanych w pamięci operacyjnej. Kolejnym przykładem są operacje odczytu i zapisu do pliku, które składają się z wielu kontynuacji. Te operacje wymagają, aby plik istniał do momentu zakończenia przetwarzania operacji.

Cały mechanizm synchronizacji nie powinien ograniczać współbieżności w większym stopniu niż to konieczne.

Rozwiązanie

W zależności od operacji potrzeba założyć blokadę na i-węzeł lub wpis katalogowy, a czasami na wiele z tych rzeczy jednocześnie, przykłady obrazujące dlaczego potrzebujemy takie rodzaje blokad znajduje się w 2.2. Aby uniknąć zakleszczeń, każda operacja musi stwierdzić, jakich blokad potrzebuje i zlecić założenie ich wszystkich podsystemowi zarządzającemu blokadami, który potrafi zrobić to w bezpieczny sposób. Jednocześnie sytuacja, w której operacja założyła wcześniej jakieś blokady i zleca założenie nowych przed zwolnieniem starych, jest niedopuszczalna, gdyż to może prowadzić do zakleszczenia.

Niech x oznacza numer i-węzła. Wówczas (x) oznacza blokadę dla tego i-węzła, a (x, w) oznacza blokadę dla wpisu katalogowego w powiązanego z tym i-węzłem. Wprowadźmy po-

rządek na wszystkich blokadach:

$$\begin{aligned}(x) < (y) &\Leftrightarrow x < y \\ (x) < (y, w) &\Leftrightarrow x \leq y \\ (x, w) < (y) &\Leftrightarrow x < y \\ (x, w) < (y, v) &\Leftrightarrow x < y \vee (x = y \wedge w < v)\end{aligned}$$

Jest to porządek leksykograficzny, ale tak naprawdę mógłby to być dowolny porządek, dopóki byłby liniowy.

Gdy potrzeba założyć kilka blokad jednocześnie, to podsystem zarządzający blokadami zakłada je w opisanym porządku liniowym. W ten sposób można zakładać dowolną ilość blokad jednocześnie, bez ryzyka zakleszczenia. Więcej o tym rozwiązaniu można przeczytać w [18].

Używamy blokad typu czytelnicy-pisarze. Wynika to z tego, że niektóre operacje wymagają, aby żadna inna operacja nie używała danego obiektu, np. operacje tworzące lub usuwające i-węzeł lub wpis katalogowy. Natomiast pozostałe operacje, które jedynie potrzebują, aby obiekt nie zniknął w czasie swojej realizacji, zakładają blokadę współdzieloną.

Przykłady

Tworzenie pliku wymaga założenia współdzielonej blokady na i-węzeł katalogu w którym jest tworzony i wyłącznej blokady na dodawany wpis katalogowy. Gdyby nie była założona blokada na katalog i byłby on pusty, to inna operacja mogłaby w trakcie tworzenia pliku usunąć katalog. Wtedy dokończenie tworzenia pliku doprowadziłoby do uszkodzenia metadanych. Gdyby nie użyć blokady na wpis katalogowy, to można by było uruchomić na raz dwie operacje tworzenia pliku o tej samej ścieżce i obie mogłyby się powieść. Taka sytuacja w zależności od implementacji może doprowadzić do tego, że jeden plik będzie podpięty w katalogu, a drugi będzie nieprzypisany do żadnego katalogu.

Odczepienie (*ang.* unlink) pliku wymaga założenie wyłącznej blokady na wpisie katalogowym. Tutaj blokowanie i-węzła jest zbędne, ponieważ jak uda nim się zablokować wpis katalogowy, to nikt nie będzie mógł usunąć katalogu w którym się on znajduje, ponieważ usuwać można jedynie puste katalogi.

Zapis do lub odczyt z pliku wymaga założenia współdzielonej blokady na i-węźle czytanego pliku.

2.3. Realizacja odczytów i zapisów

Dane pliku mogą być trzymane w trzech różnych formach – jako mały, średni lub duży zapis. Małe zapisy cechują się tym, że są wielkości co najwyżej kilku kilobajtów, duże zapisy odpowiadają zawsze pewnemu klastrowi i zajmują całą jego przestrzeń, a średnie zapisy są dłuższe od małych zapisów, ale krótsze od dużych. Jeśli operacja zapisu jest większa niż rozmiar jednego klastra, to jest dzielona na mniejsze części, gdzie każda należy do jednego z wymienionych typów.

Dane zapisu trafiają do miejsca zależnego od ich typu – małe zapisy na koniec logu metadanych, średnie na koniec logu danych, a duże do nowego klastra na dysku. Wszystkie małe zapisy trzymane są też przez cały czas w pamięci operacyjnej.

Metadane modyfikowanego pliku trafiają do dwóch miejsc. Pierwszym są struktury plików w pamięci zawierające informacje o lokalizacji każdego fragmentu danych pliku (adresy w pamięci lub pozycje na dysku). Modyfikacja tych struktur może wiązać się z usunięciem lub

aktualizacją informacji o poprzednich zapisach obejmujących modyfikowany fragment pliku. Drugim miejscem jest log metadanych. Log metadanych jest początkowo tworzony w pamięci, aż do osiągnięcia wielkości jednego klastra. Wtedy jest zrzucany na dysk i zwalniany z pamięci.

Zauważmy, że w przeciwieństwie do innych systemów plików, modyfikacje danych w pliku tak na prawdę nie modyfikują tych danych na dysku. Dane i metadane zawsze trafiają w nowe miejsce i jedynie metadane w pamięci są aktualizowane, aby zawierały informacje o nowej pozycji zmodyfikowanego fragmentu pliku. Taka strategia może spowodować, że modyfikacje pliku "rozproszą" jego dane po wielu klastrach na dysku.

Operacje odczytu znajdują w strukturach w pamięci lokalizacje zapisanych fragmentów danych pliku, a następnie wykonują na nich operacje zależne od ich typu: albo odczytują dane z dysku, albo przekopiowują je z pamięci.

Znaczącymi zaletami takiego schematu są:

- szybki dostęp do danych w małych plikach – odczyt i zapis jest wykonywany w pamięci operacyjnej, co powoduje bardzo dużą efektywność tych operacji.
- dane z dużych zapisów są zapisywane na dysku w spójnych kawałkach o wielkości klastra, a takie operacje dyski potrafią obsługiwać z dużą przepustowością.
- przy braku modyfikacji dane plików nie będą w dużym stopniu rozproszone po dysku, przez co operacje odczytu będą mogły działać z większą przepustowością.

Natomiast wyszczególniają się pewne wady:

- wiele małych zapisów może spowodować zajęcie całej pamięci operacyjnej.
- modyfikacje zapisanych danych powodują wspomniane wcześniej "rozproszenie" danych po dysku, co ogranicza efektywność operacji odczytu, które zamiast odczytywać dane ze spójnych przedziałów, muszą odczytywać je z wielu kawałków porozrzucanych po całym dysku.

Z tych powodów SeastarFS nie jest dobrym systemem plików w przypadku ogólnym. Jednak dla bazy danych Scylla, może okazać się wyjątkowo dobry.

Aby zrozumieć powód użycia tego schematu w SeastarFS, należy najpierw spojrzeć na rodzaje operacji wykonywanych w bazie danych Scylla, pod którą system plików był optymalizowany. Gdy dane tabel są modyfikowane, Scylla przechowuje informacje o tym w dwóch miejscach: memtable i commitlog. Commitlog jest plikiem na dysku, do którego dane są jedynie dopisywane na końcu w niewielkich kawałkach. Memtable jest strukturą w pamięci zawierającą dane tabel. Kiedy ta struktura osiągnie odpowiedni rozmiar (około 160MB) zostaje zapisana na dysk jako plik SSTable. Ponieważ dane w tych plikach nigdy nie są modyfikowane, to po pewnym czasie duża ich część staje się nieaktualna marnując miejsce na dysku. Oprócz tego po pewnym czasie wyszukiwane dane stają się porozrzucane po wielu plikach, co redukuje efektywność odczytu, który zamiast odczytać jeden plik, musi odczytać kilka. Aby przeciwdziałać tym problemom, co jakiś czas zostaje uruchomiona kompakcja, która odczytuje kilka SSTable i zapisuje nowe pliki SSTable z jedynie aktualnymi danymi, więcej informacji o kompakcji w Scylli można znaleźć w [14].

Operacje na commitlogu cechują się tym, że dane są dodawane na końcu pliku oraz są wielkości małych lub średnich zapisów. Natomiast dane SSTable są zawsze odczytywane i zapisywane w całości w długich spójnych kawałkach. Oprócz tego w żadnym z tych plików nie występują operacje modyfikacji danych, jedynie pojawiają się czasami usunięcia całych plików (usunięcie starych SSTable po kompakcji). W takim przypadku wspomniane wcześniej wady realizacji operacji odczytów i zapisów nie występują.

2.4. Podział na rdzenie

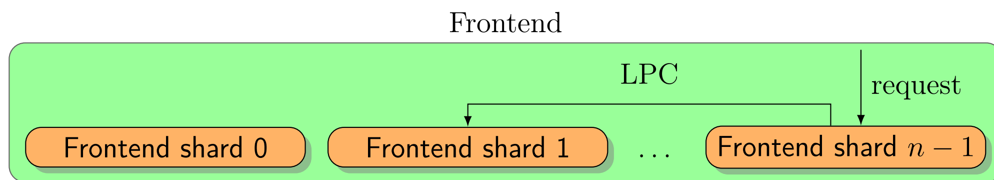
Podział na rdzenie jest strategią do skalowania pionowego aplikacji, co w naszym przypadku oznacza podział procesora, dysku i pamięci operacyjnej. SeastarFS, zgodnie z ideą braku współdzielenia zasobów [17], jest dzielony na niezależne instancje, tak jak jest to wspomniane we wstępie do rozdziału 2.

2.4.1. Podział dysku

Cały dysk jest podzielony na równe części (domeny) odpowiadające instancjom systemu plików, czyli każda instancja jest odpowiedzialna za swoją domenę. Głównym zadaniem instancji systemu plików jest operowanie w swojej domenie. Gdy instancja otrzyma zapytanie dotyczące innej domeny, przekazuje je innej instancji, opis tego znajduje się w podsekcji 2.4.2. Działanie wyłącznie w swojej domenie pomaga utrzymywać bardzo dobry poziom zrównoleglenia całego systemu, ponieważ pozwala to uniknąć drogiej komunikacji między instancjami.

2.4.2. Operacje na instancji

Jeżeli zlecona do instancji operacja (`open`, `create`, `remove`) nie należy do zakresu jej domeny, to wykonywane jest przekierowanie zlecenia do właściwej instancji za pośrednictwem LPC (*ang.* Local Procedure Call) [15]. Intensywne operacje na plikach nienależących do odpowiedniej domeny instancji wiążą się z dodatkowym narzutem w postaci LPC, natomiast operowanie na plikach w swojej domenie pozwala tego uniknąć. Opisany schemat przedstawia rysunek 2.2.



Rysunek 2.2: Podział systemu plików na instancje oraz ewentualne przekazywanie zlecenia za pomocą LPC.

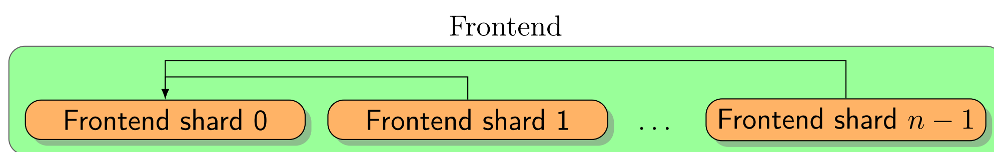
2.4.3. Współdzielony katalog główny

W związku z tym, że cały dysk jest podzielony na niezależne części i system plików jest rozproszony na rdzeniach, występuje problem z utrzymywaniem spójności niektórych danych między rdzeniami. Takim właśnie kluczowym miejscem synchronizacji jest katalog główny (*ang.* root directory) systemu plików. Każda instancja systemu posiada własny katalog główny, ale jest przechowywana też zbiorcza informacja o katalogach głównych każdej instancji.

Źródło danych

Aby mieć spójną informację o globalnym katalogu głównym systemu plików, do replikacji danych pomiędzy instancjami został użyty asymetryczny model komunikacji mistrz-niewolnik (*ang.* master-slave), gdzie jedna z instancji występuje w postaci wiarygodnego źródła danych, a pozostałe instancje są z nią synchronizowane. Opisany schemat przedstawia rysunek 2.3.

Podczas uruchamiania systemu plików, każda instancja przesyła do Mistrza informacje o stanie katalogu głównego w swojej domenie. Zakłada się, że informacja po wyłączeniu systemu plików była poprawna i Mistrz nie pozwolił na dokonanie błędnych operacji w katalogu głównym. Cały system nie zakończy uruchamiania, dopóki mistrz nie zbierze informacji o globalnym stanie katalogu głównego. Mistrzem staje się ta instancja, która zainicjowała uruchomienie całego systemu. Rola Mistrza nie jest przekazywana w trakcie działania systemu plików.



Rysunek 2.3: Schemat przedstawia podział systemu plików na instancje oraz synchronizację pomiędzy nimi za pośrednictwem wybranego Mistrza.

Zmiany w katalogu głównym

Jeżeli instancja zamierza dokonać zmian w katalogu głównym, to wysyła zapytanie do Mistrza. W przypadku zgody, mistrz rozgłasza informację o tym wszystkim instancjom, a one zapisują te dane, wpp. odpowiada odmową. Dzięki temu system unika pojawienia się duplikatów w katalogu głównym.

W celu uniknięcia ciągłej komunikacji z Mistrzem, użytkownik powinien stworzyć w katalogu głównym lokalny katalog, a następnie wykonywać w nim wszystkie operacje. Dzięki temu komunikacja z Mistrzem nastąpi jedynie przy tworzeniu katalogu w katalogu głównym, a wszystkie pozostałe operacje będą wykonywane lokalnie.

Struktura danych

Informacje o katalogu głównym rozgłaszane przez Mistrza są przechowywane w słowniku z nazwą pliku/katalogu jako kluczem oraz identyfikatorem instancji jako wartością. Instancja przed wykonaniem operacji sprawdza w słowniku, na której domenie powinna być zrealizowana.

2.4.4. Operacje na pliku

Każdy otwierany plik jest dowiązywany do właściwej domeny i przy otwieraniu pliku jest podejmowana decyzja czy będzie potrzebny LPC do odwołania się do nielokalnej domeny podczas każdej operacji (`read`, `write`, `truncate`, `flush`, `close`, `stat`, `size`) na pliku. Otworzone pliki w domenie, która odpowiada lokalnej instancji, nie używają LPC do wykonania operacji.

2.5. Kompakcja

Ponieważ SeastarFS należy do klasy systemów plików o strukturze logu, po dopisaniu nowych danych, niektóre fragmenty wcześniej zapisanych danych mogą stać się nieaktualne. Dodatkowo jednym z założeń było to, że większość operacji będzie realizowana na spójnych przedziałach o wielkości klastra, co utrudnia odzyskiwanie z nich nieużywanego miejsca.

Kompakcja jest mechanizmem odzyskiwania miejsca w SeastarFS. Jest to mechanizm w pewnym stopniu wzorowany na mechanizmach odśmiecania pamięci (*ang.* garbage collection). W czasie działania, system plików korzysta z różnego rodzaju heurystyk (opisanych w podsekcji 2.5.1) w celu zdecydowania kiedy i które klastry mają zostać odśmiecone. Następnie podane klastry są odczytywane i fragmenty zawierające aktualne dane są przepisywane do nowych klastrów, a pozostałe fragmenty są pomijane. W ten sposób kompakcja odzyskuje różnicę liczby klastrów potrzebnych na przechowywanie danych przed rozpoczęciem kompakcji i liczby klastrów potrzebnych po jej zakończeniu.

W podrozdziałach opisane są wspomniane heurystyki decydujące o uruchomieniu kompakcji oraz sam sposób jej realizacji na wyznaczonych klastrach.

2.5.1. Wybór klastrów do kompakcji

Kompakcja wiąże się z koniecznością wykonania dodatkowych operacji na dysku. Może to skutkować wolniejszym działaniem systemu plików podczas wykonywania operacji zlecanych przez użytkownika. Z tego powodu kompakcja powinna być wykonywana jak najrzadziej, wciąż jednak wystarczająco często, aby jej brak nie powodował problemów z brakiem miejsca. Aby ustalić moment, w którym należy przeprowadzić kompakcję, brane są pod uwagę informacje o stanie zapełnienia każdego klastra aktualnymi danymi.

Przy uruchamianiu systemu plików podawane są parametry k i M. Gdy w pewnym momencie stosunek aktualnych danych w klastrze do rozmiaru klastra spadnie poniżej k, dany klaster jest oznaczany jako nadający się do kompakcji. Parametr M oznacza maksymalną wielkość danych, które mogą być odczytane i równocześnie przechowywane pamięci w trakcie kompakcji. Oznaczmy liczbę klastrów gotowych do kompakcji w danym momencie przez X. Gdy zajdzie nierówność $(X + 1)k \geq M$, to jest uruchamiana kompakcja na klastrach nadających się do kompakcji. Jednocześnie te klastry zostają usunięte z grupy klastrów nadających się do kompakcji. Takie podejście pozwala ustawić, jak duży zysk będzie uzyskiwać każda uruchamiana kompakcja.

Dla przykładu ustawiając wartość k na 0.1 oraz M na rozmiar jednego klastra kompakcja będzie się uruchamiała dla 10 klastrów, każdy o zajętości co najwyżej 10%, a dane po kompakcji będę zajmować co najwyżej jeden pełny klaster. Ciekawą konfiguracją jest ustawienie parametrów k i M na 0. To spowoduje, że klastry będą kompaktowane tylko wtedy, kiedy nie będą miały aktualnych danych, co będzie bardzo efektywne, ponieważ nie będzie to wymagało odczytania żadnych danych z dysku. Wada jest jednak taka, że kompakcja będzie mało skuteczna.

Testowaliśmy też modyfikację wspomnianej strategii. Parametr k był ustawiany dynamicznie w czasie działania systemu i był uzależniony od ilości wolnego miejsca na dysku, konkretniej k było automatycznie ustawiane na wartość stosunku wolnych do wszystkich dostępnych klastrów. Aby uniknąć problemów przy małych ilościach wolnego miejsca, gdzie z taką strategią wykonywalibyśmy dużo mało efektywnych kompakcji, wartość k była ograniczana przez pewną wartość podawaną przy uruchamianiu systemu plików. Takie podejście osiągało lepsze wyniki od wspomnianej wcześniej kompakcji ze stałą wartością k. Porównanie wyników tych strategii umieściliśmy w podsekcji 3.1.2.

Innym spotykanym podejściem uwzględnianie wieku różnych fragmentów danych w klastrze [12]. Wówczas, aby zagwarantować wysokie zapełnienie wielu klastrów przewiduje się na jego podstawie, jakie dane mają małą szansę na usunięcie i przepisuje się je do nowych klastrów. Wtedy stopień zapełnienia tychże klastrów utrzymuje się przez dłuższy czas na wysokim poziomie, a pozostałe są kompaktowane w szybszym tempie. Przekłada się to na lepsze spożytkowanie miejsca na dysku.

2.5.2. Realizacja kompakcji

Schemat kompakcji danych wygląda w następujący sposób:

1. Odczytanie aktualnych danych ze wskazanych klastrów do pamięci.
2. Zaalokowanie nowych klastrów do których zostaną przepisane kompaktowane dane.
3. Ustalenie, jak kompaktowane dane zostaną ułożone w nowych klastrach.
4. Zapisanie nowych klastrów na dysk z jednoczesną aktualizacją metadanych w pamięci o nowe pozycje danych.
5. Zwolnienie starych klastrów.

Przy odczytywaniu danych z klastrów trzeba wziąć pod uwagę, jaki procent klastra jest zajęty przez aktualne dane oraz liczbę fragmentów odczytywanych danych z klastra. Wynika to z tego, że dyski potrafią lepiej obsługiwać odczytywanie długich spójnych kawałków dysku, niż wielu porozrzucanych. Z tego powodu zostały rozważone dwa sposoby realizacji tego zadania:

1. Odczytywanie jedynie tych fragmentów klastrów, które zawierają aktualne dane.
2. Odczytywanie zawsze całych klastrów, a następnie wyodrębnienie tych fragmentów, które zawierają aktualne dane.

Po testach wykonanych aplikacją `cluster_reading_perf` umieszczonej na płycie CD (lista materiałów na płycie CD znajduje się w dodatku B) można było zauważyć, że pierwsze rozwiązanie osiągało lepsze wyniki w przypadku małej liczby fragmentów oraz małej procentowej zajętości odczytywanych danych z klastra. Drugie rozwiązanie okazywało się lepsze kiedy odczytywanych fragmentów było dużo i klaster miał w większości aktualne dane. Na wyniki miał wpływ też sam zrąb Seastar, który w przypadku wielu fragmentów musiał przetworzyć wiele operacji wejścia-wyjścia i wszystkie dodatkowe akcje z tym związane, tj. alokacje pamięci, zarządzanie asynchronicznością, itp. Ostatecznie w kodzie programu zostało użyte pierwsze rozwiązanie, jednak prawdopodobnie lepsze byłoby podejście "hybrydowe" polegające na użyciu pierwszego lub drugiego rozwiązania na podstawie szacowań, które będzie bardziej efektywne.

Klastry przeznaczone na kompakcję są pobierane z tej samej puli klastrów, z której brane są klastry na dane w operacjach zapisu. Kompakcja przed zwolnieniem kompaktowanych klastrów potrzebuje najpierw zaalokować klastry, do których zostaną przepisane dane. W przypadku, kiedy takich klastrów nie da się zaalokować, kompakcja nie jest w stanie się zakończyć i zwolnić miejsce. Alternatywne i bardziej bezpieczne rozwiązanie byłoby wydzielenie pewnej puli klastrów przeznaczonych tylko na kompakcję. Takie podejście eliminuje wspomniany problem, ponieważ alokacja klastrów na dane nie ma wpływu na klastry przeznaczone dla kompakcji. Kompakcja zawsze będzie miała do dyspozycji klastry, które może użyć do zapisania kompaktowanych danych.

Odczytane fragmenty danych są zapisywane w kolejnych zaalokowanych klastrach. Jeśli któryś fragment nie mieści się w aktualnie zapełnianym klastrze, zostaje on podzielony na dwa fragmenty i drugi trafia do kolejnego z zaalokowanych klastrów. Takie rozwiązanie zapewnia najbardziej optymalną kompakcję danych, ponieważ wszystkie klastry (poza być może ostatnim) są całkowicie zapełnione. Ulepszone rozwiązanie mogłoby grupować odczytane fragmenty po plikach, z których te fragmenty pochodzą i zapisywać fragmenty z tego samego pliku obok siebie. To pozwoliłoby na pewną defragmentację plików w czasie kompakcji, poprawiając później efektywność operacji odczytów.

Rozdział 3

Testy efektywności i analiza wyników

W celu zweryfikowania efektywności przedstawionego systemu plików przeprowadzono szereg testów efektywnościowych. W rozdziale 3.1 przedstawiono wyniki testów przeprowadzonych z różnymi parametrami SeastarFS. Niektórych wyników przedstawionych w tym rozdziale użyto do wyboru parametrów przy przeprowadzaniu testów porównawczych naszego systemu plików z systemami XFS i ext4, których wyniki przedstawiono w rozdziale 3.2.

Do symulacji działania systemu plików w dłuższym przedziale czasu stworzono aplikację `fs_perf`. Jej działanie polega na zmierzeniu czasu, w jakim dany system plików wykonuje ciąg losowych zapisów i odczytów określonych przez podane parametry.

W analizowanych przypadkach testowych ciąg ten kończy się po zapisaniu dużej ilości danych do dysku, stałej dla wszystkich testów. Pozwala to na porównywanie wyników z różnych testów, jak również ogranicza wpływ losowości na uzyskany wynik. Aby upodobnić sprawdzany schemat działania systemu do schematu działania bazy danych Scylla, wszystkie zapisy zaczynają się na końcach plików. Jeśli nie napisano inaczej, w testach operacje zapisu stanowią 80% wszystkich operacji, małe zapisy stanowią 10% wszystkich zapisów, a małe odczyty stanowią 10% wszystkich odczytów. W każdej instancji małe operacje są wykonywane na losowym spośród 16 przeznaczonych do tego plików, a duże na losowym spośród innych 16 plików. Rozmiar małej operacji to losowa wartość z zakresu [8KiB, 128KiB], a dużej – z zakresu [16MiB, 20MiB]. Gdy stopień zapełnienia dysku aktualnymi danymi przekroczy 50%, wykonywane zostają operacje zmniejszenia plików do momentu, gdy wartość ta spadnie poniżej 25%. Zmniejszanie plików działa na podobnej zasadzie jak operacje odczytu i zapisu. Najpierw losowany jest plik, a następnie losowana jest ilość danych, która zostanie usunięta z końca wylosowanego pliku. Prawdopodobieństwa i przedziały ilości usuwanych danych są takie same jak w przypadku operacji odczytu i zapisu.

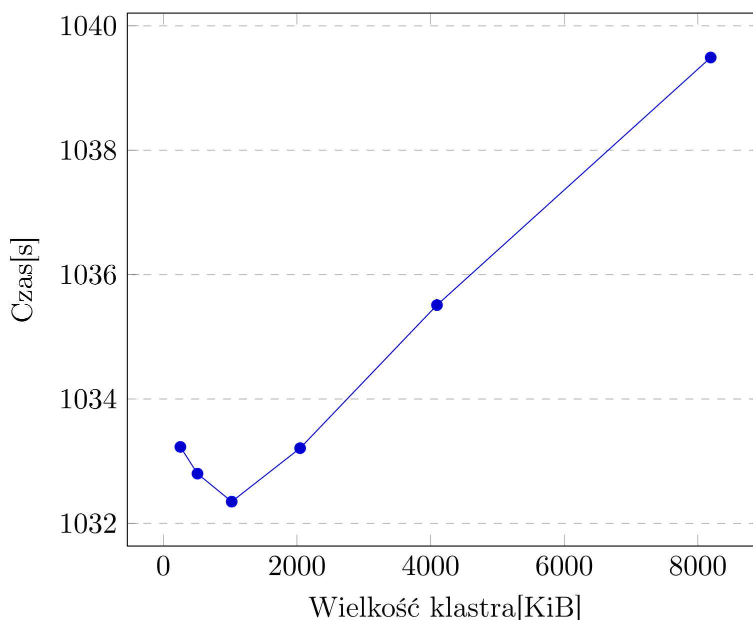
Przed każdą próbą dysk jest zapełniany wedle opisanego schematu z różnicą polegającą na tym, że w trakcie zapełniania nie wykonuje się odczytów. Etap ten nie jest uwzględniany w wynikowym czasie.

Do mierzenia opóźnienia SeastarFS stworzono aplikację `sfs_io_tester` na podstawie już istniejącej aplikacji `io_tester`. Działają one w analogiczny sposób – wykonują pewne ustalone działania na dysku na wszystkich logicznych rdzeniach procesora z pewnymi ustawieniami, takimi jak poziom zrównoleglenia czy rozmiar pojedynczej operacji.

3.1. Porównanie parametrów SeastarFS

W tym podrozdziale przedstawiono wyniki porównania różnych parametrów systemu plików SestarFS.

3.1.1. Wielkość klastra



Rysunek 3.1: Czas wykonania testu dla różnych wielkości klastra

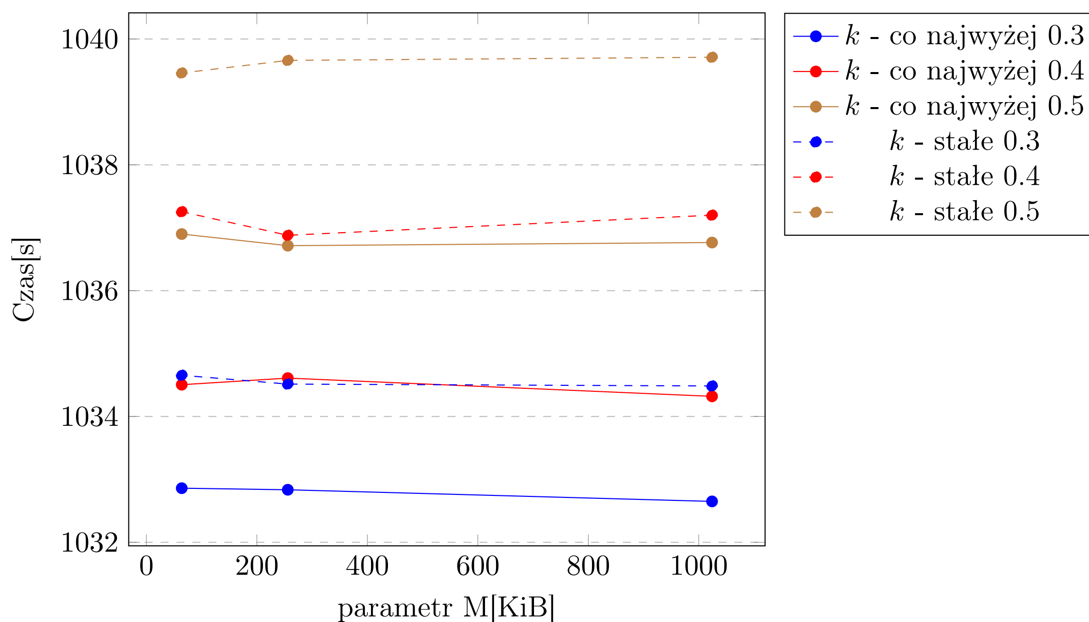
Z wyników widocznych na wykresie 3.1 wynika, że rozmiar klastra ma znaczenie dla szybkości systemu.

Nie powinien być on za mały, ponieważ dysk wydajniej wykonuje większe zapisy, a zapisy są dzielone na mniejsze wielkości co najwyżej klastra. Większa liczba mniejszych klastrów wiąże się również z większym narzutem czasowym występującym przy zarządzaniu nimi.

W naszym teście system działa wolniej gdy klastry są za duże. Prawdopodobnie jest to spowodowane tym, że przy większych redukcjach rozmiaru pliku, które często występują w testowanym scenariuszu, większa część usuwanych danych zajmie całe klastry, jeśli są one mniejsze, a puste klastry można odzyskać bez przepisywania danych, co przekłada się na mniejszą liczbę dodatkowych operacji wejścia-wyjścia w trakcie kompakcji. Jednoznaczne ustalenie przyczyny wolniejszego działania systemu z większymi klastrami wymagałoby obszerniejszych badań.

W przypadku naszego dysku optymalnym rozmiarem okazał się 1MiB. Tyle też jest równa wielkość klastra w pozostałych testach.

3.1.2. Strategie kompakcji



Rysunek 3.2: Czas wykonania testu dla różnych kryteriów wykonywania kompakcji

Aby zmierzyć wpływ kompakcji na czas wykonania testu, porównaliśmy różne wartości parametrów k oraz M. Porównaliśmy wersje, w których k wzrasta wraz z zapełnieniem dysku aż do ustalonej wartości oraz takie, w których jest on stały. Parametry k, M i strategie kompakcji zostały opisane w rozdziale 2.5.1).

Ze zmierzonych czasów pokazanych na wykresie 3.2 wynika, że kompakcja przebiega tym szybciej, im mniejszy jest parametr k, czyli im mniej zapełnione są kompaktowane klastry. System działa szybciej, gdy k jest zależny od zapełnienia dysku. Wpływ maksymalnej ilości równocześnie kompaktowanych danych jest minimalny.

Zmniejszanie parametru k wpływa pozytywnie na czas wykonywania testu, ponieważ im jest on mniejszy, tym rzadziej przeprowadzana jest kompakcja, a gdy do niej dochodzi, odzyskiwana jest większa liczba klastrów.

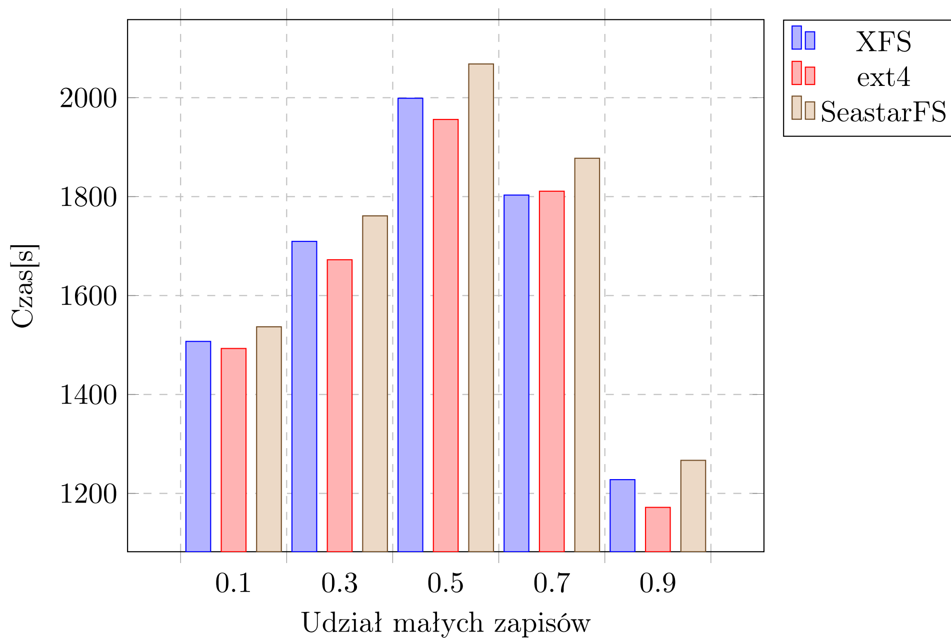
W pozostałych testach parametr k jest równy 0.3 i jest niezależny od stopnia zapełnienia dysku, a maksymalna wielkość kompaktowanych danych wynosi 256MiB.

3.2. Porównanie z XFS i ext4

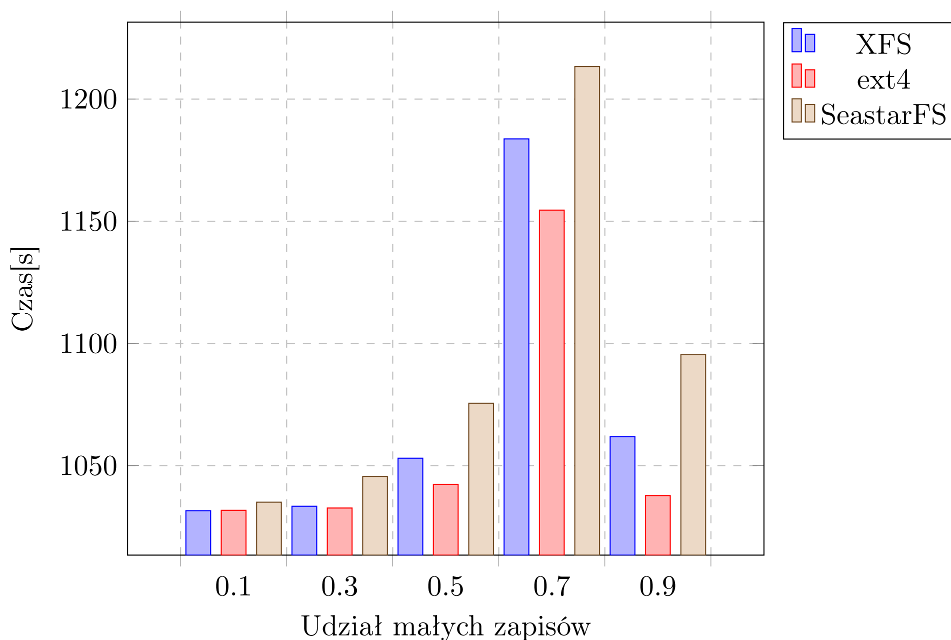
W tym podrozdziale zostały przedstawione wyniki porównania systemów plików SeastarFS z XFS i ext4.

3.2.1. Porównanie w symulacji

Aplikacja `fs_perf` pozwala na testowanie również innych systemów plików na tym samym scenariuszu. W tym podrozdziale przedstawione są porównania wyników SeastarFS z wynikami innych systemów.

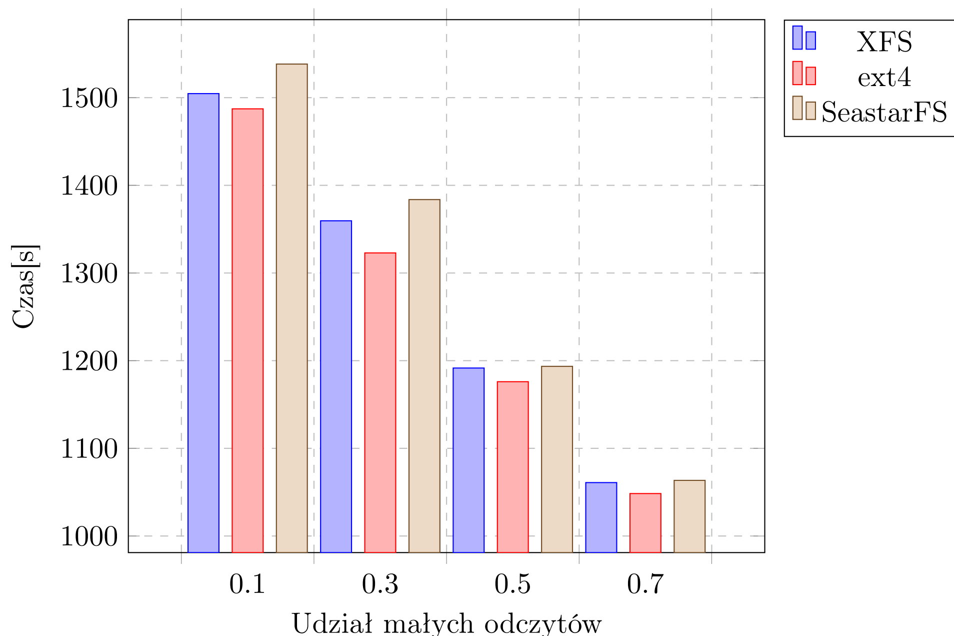


Rysunek 3.3: Czas wykonania testu dla różnych proporcji małych do wszystkich zapisów przy zapisach stanowiących 40% wszystkich operacji



Rysunek 3.4: Czas wykonania testu dla różnych proporcji małych do wszystkich zapisów przy zapisach stanowiących 80% wszystkich operacji

Na wykresach 3.3 i 3.4 możemy zauważyć, że różnice w czasie wykonania testu przez SeastarFS i systemy XFS oraz ext4 są najmniejsze, gdy małe zapisy stanowią najmniejszą część zapisów. Różnica wynosi wówczas mniej niż 3%. W najgorszym przypadku różnica ta nie przekracza 9%.



Rysunek 3.5: Czas wykonania testu dla różnych proporcji małych do wszystkich odczytów przy odczytach stanowiących 60% wszystkich operacji

Jak widać na wykresie 3.5, wielkość odczytów nie ma większego znaczenia na szybkość naszego systemu. Różnica między czasem działania testu na systemie SeastarFS, a na systemie XFS nie przekracza 2.5%, a między systemami SeastarFS i ext4 - 5%. Udział małych odczytów ma wpływ na ilość odczytanych danych, zatem nie można porównać bezwzględnych wyników czasowych z różnymi wartościami udziału małych odczytów.

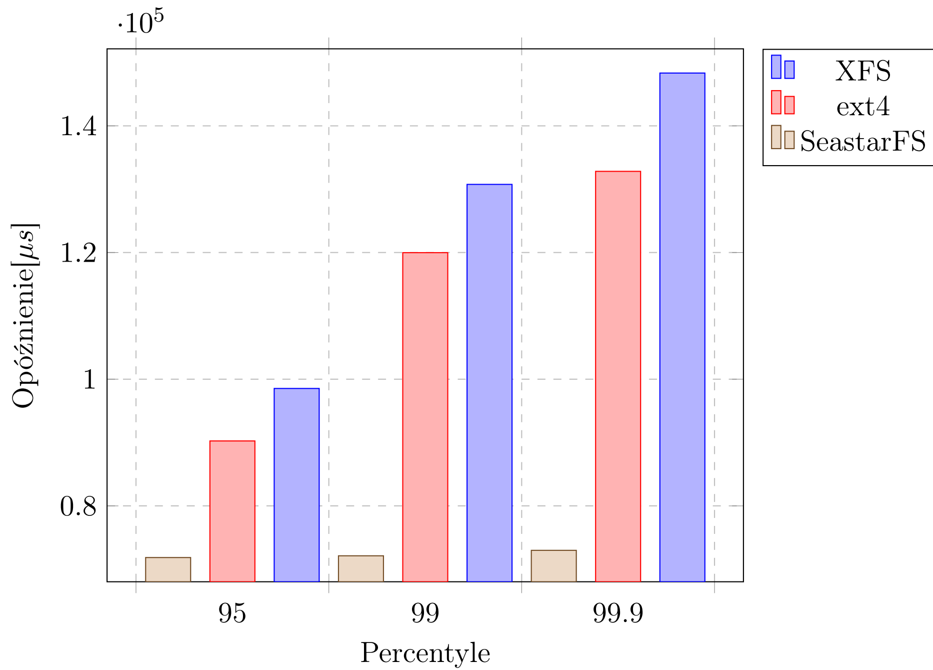
3.2.2. Porównanie opóźnienia

Porównanie opóźnienia zostało zrobione z rozmiarem klastra 4MiB w SeastarFS, wykonanie każdej akcji podczas badania trwało 10 sekund na 36 logicznych rdzeniach z poziomem zrównoleglenia 10 w zrębie Seastar. Każda instancja działała na jednym pliku z rozmiarem 1GiB. Głównymi wykonywanymi operacjami w bazie danych Scylla są duże sekwencyjne zapisy, duże wstawiania na koniec i duże sekwencyjne odczyty, więc takie działania zostały przebadane.

Na przedstawionych wykresach opóźnienie jest mierzone w mikrosekundach (μs), a dane są zbierane w percentylach. W tym przypadku wartość X dla percentylu P oznacza, że P procent operacji zakończyło się z czasem co najwyżej X, a $100-P$ procent z czasem większym niż X. W testach przedstawiliśmy wartości dla percentyli 95, 99 i 99.9.

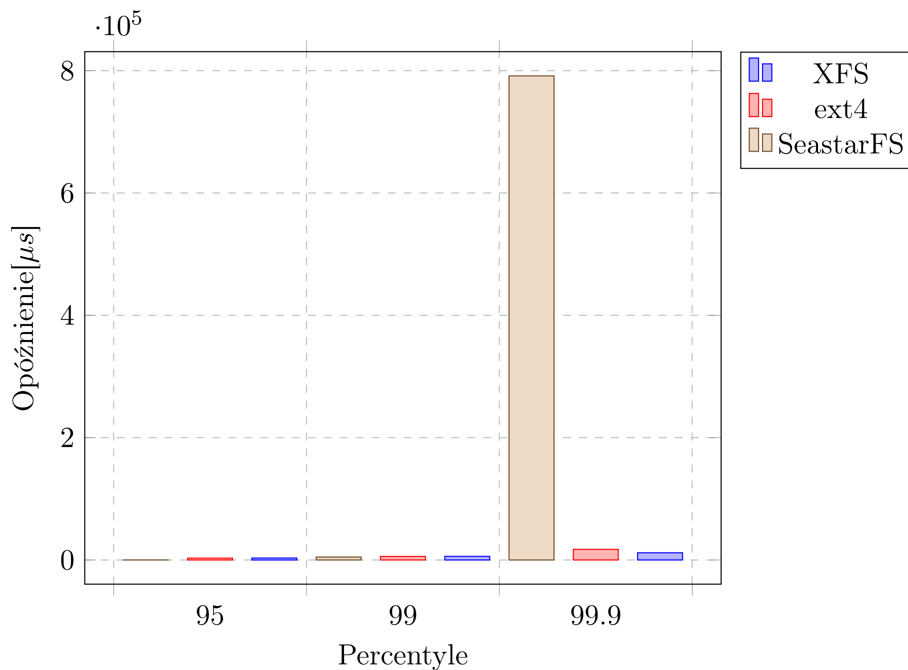
Zapisy

Do badania zapisów został wybrany rozmiar 128 KiB pojedynczej operacji. Do pokazania nieefektywności SeastarFS z operacjami na małych danych został wybrany rozmiar 4 KiB.



Rysunek 3.6: Wyniki opóźnienia sekwencyjnych zapisów po 128 KiB

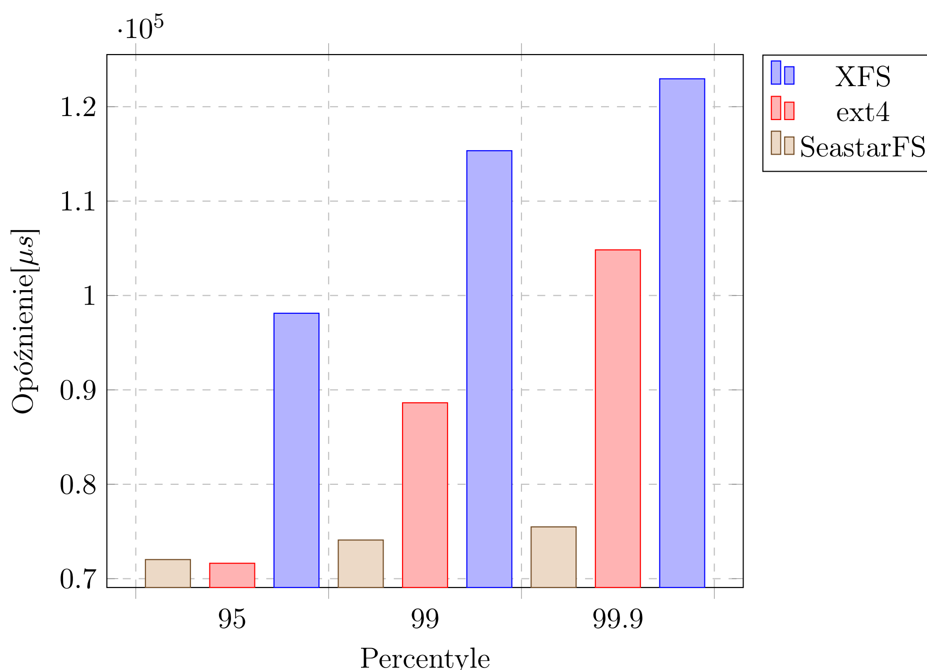
Wykonanie sekwencyjnych zapisów polega na tym, że dane do pliku są zapisywane od początku do końca i nie przekraczają początkowego rozmiaru pliku. Na rysunku 3.6 widać, że SeastarFS osiąga wyniki o 50.80% lepsze od XFS i o 45.05% od ext4 w 99.9 percentylu. W pozostałych percentylach wyniki też są lepsze, ale już mniej.



Rysunek 3.7: Wyniki opóźnienia sekwencyjnych zapisów po 4 KiB

Na rysunku 3.7 widać, że SeastarFS nie radzi sobie z małymi sekwencyjnymi zapisami,

osiągając wyniki nawet o kilkadziesiąt razy gorsze od XFS i ext4 w 99.9 percentylu, ale już w 95 percentylu osiąga odpowiednio o 99.55% i 99.52% lepsze wyniki. Podejrzewamy, że dzieje się tak dlatego, że w SeastarFS małe zapisy są zbierane w pamięci i później są zrzucane na dysk, a wykonanie destruktorów i zwolnienie pamięci wykonuje się synchronicznie w C++, co znacznie zwiększa opóźnienie.



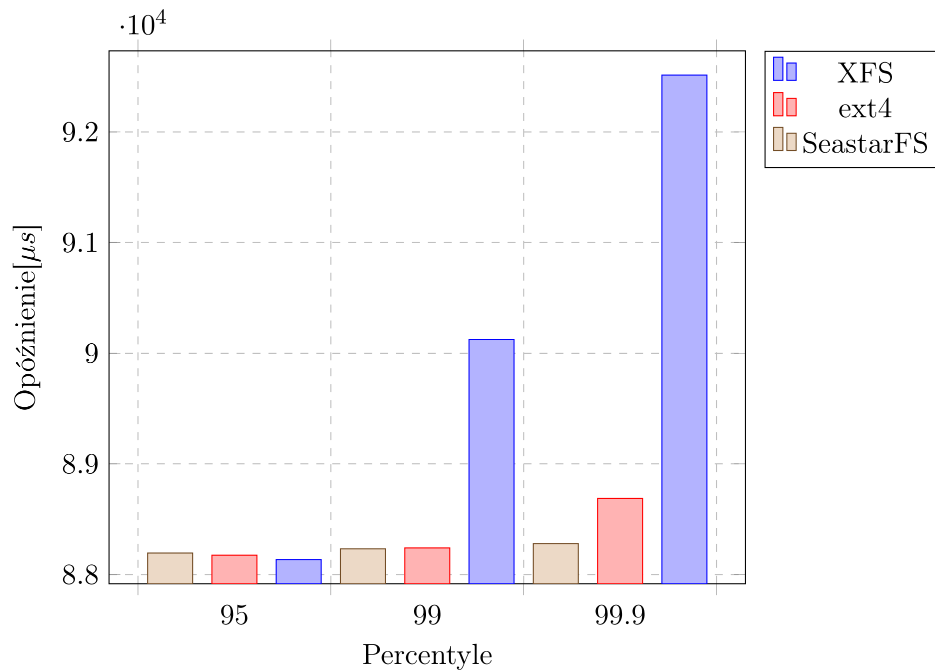
Rysunek 3.8: Wstawianie na koniec po 128 KiB

Wstawianie na koniec, w odróżnieniu od sekwencyjnych zapisów, dodaje dane tylko na koniec pliku zaczynając od pliku o rozmiarze 1GiB. Na rysunku 3.8 widać, że SeastarFS dobrze sobie radzi w 99.9 percentylu i osiąga wyniki o 38.61% lepsze od XFS i o 27.99% od ext4. W 95 percentylu SeastarFS osiąga gorszy wynik od ext4, ale tylko o 0.55%.

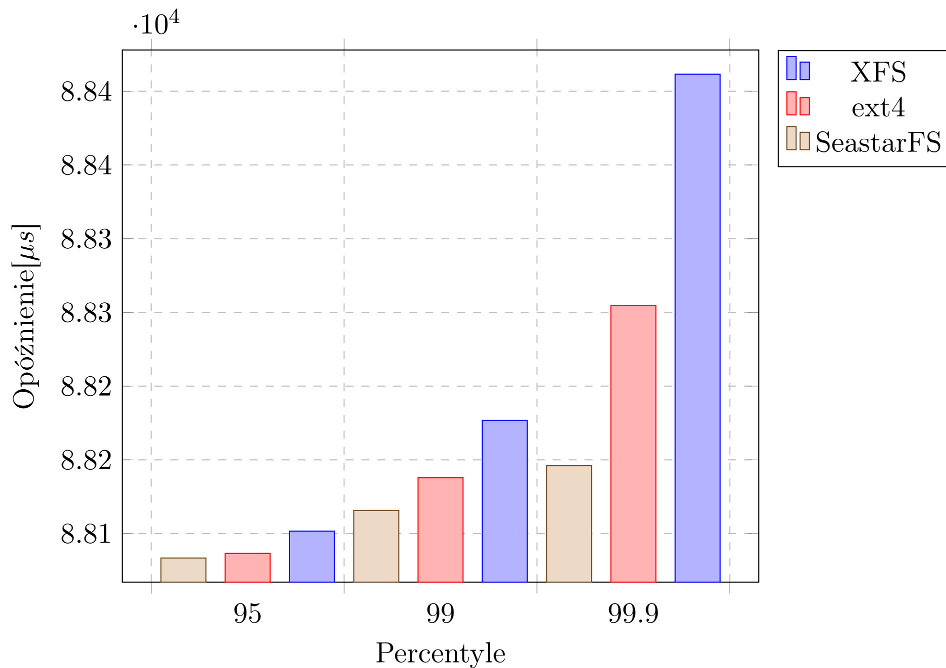
Odczyty

Do badania odczytów został wybrany rozmiar 256 KiB pojedynczej operacji.

Sekwencyjne odczyty są wykonywane od początku do końca pliku. Na rysunku 3.9 widać, że SeastarFS dobrze radzi sobie w 99.9 percentylu i osiąga wyniki o 4.58% lepsze od XFS i o 0.46% od ext4. W 95 percentylu ma wynik gorszy od ext4 o 0.02% i od XFS o 0.07%.



Rysunek 3.9: Wyniki opóźnienia sekwencyjnych odczytów po 256 KiB



Rysunek 3.10: Wyniki opóźnienia losowych odczytów po 256KiB

Losowe odczyty są wykonywane w losowym miejscu w przedziale pliku. Na rysunku 3.10 widać, że SeastarFS osiąga lepsze wyniki we wszystkich percentylach w porównaniu do XFS i ext4, ale ta różnica jest niewielka: od 0.00% do 0.30%.

Rozdział 4

Podsumowanie i kierunki rozwoju

W ramach tej pracy udało się zaimplementować w pełni asynchroniczny system plików w przestrzeni użytkownika zaprojektowany z myślą o specyfice operacji wykonywanych w bazie danych oraz wykonać na nim testy efektywnościowe. Oprócz tego, mieliśmy zbadać, czy taki systemu plików jest w stanie być bardziej efektywny niż rozwiązania zaimplementowane w jądrze Linuxa i jak duży zysk można osiągnąć. W testach efektywnościowych pokazaliśmy, że obecne rozwiązanie nie jest szybsze niż konkurencyjne przy testach symulujących działanie systemu przez dłuższy czas, jednak osiąga niewiele gorsze wyniki. W testach opóźnienia osiąga w większości przypadków lepsze wyniki od XFS i ext4.

Taki rezultat uznajemy za pozytywny, biorąc pod uwagę ograniczony czas na realizację projektu. Ostatecznie, ScyllaDB, która zleciła nam wykonanie projektu, jest zadowolona z wyników i prawdopodobnie będzie kontynuować badanie poruszonego w pracy zagadnienia na podstawie zaimplementowanego przez nas systemu plików.

4.1. Kierunki rozwoju

Ponieważ systemy plików są obszernym tematem, to niektórych zagadnień nie byliśmy w stanie zaimplementować na czas. W kolejnych podrozdziałach są przedstawione możliwe usprawnienia.

4.1.1. Defragmentacja danych w pliku

Obecna implementacja nie radzi sobie zadowalająco w przypadku wielokrotnej modyfikacji tych samych plików, ponieważ powoduje to fragmentację i rozrzucenie danych po całym dysku. Wynika to z faktu, że dane na dysku nigdy nie są modyfikowane, a jedynie dopisywane na koniec logu.

Fragmentację plików w systemie o strukturze logu można załagodzić dodając do systemu operację defragmentacji. Działałaby ona na podobnej zasadzie jak kompakcja – uruchamiana by była w momencie, kiedy pewne heurystyki uznają ją za stosowną. Defragmentacja odczytywałaby dane pliku i przepisywałaby je w nowe miejsce redukując fragmentację i rozproszenie danych, zwiększając tym samym efektywność odczytów.

4.1.2. Kompakcja logu metadanych

Bez kompakcji logu metadanych, system plików po pewnym czasie będzie przepełniony klastrami z metadanymi.

Kompakcję logu metadanych można zrealizować przez stworzenie kopii aktualnych metadanych i danych małych zapisów ze struktur w pamięci i zapisanie ich na dysku. Jednak zwyczajne skopiowanie metadanych może trwać długo i w tym czasie problematyczne by było zezwolenie na ich modyfikowanie. Z tego powodu podczas kopiowania praktycznie żadne inne operacje nie mogłyby być obsługiwane, co spowodowałoby znaczący wzrost opóźnień operacji. Alternatywnym sposobem tworzenia takiej kopii może być uruchomienie w tle proces bootstrappingu kompaktowanego metadatalogu. Takie rozwiązanie nie ma problemu ze wzrostem opóźnienia, ponieważ tworzenie metadanych jest rozłożone w dłuższym czasie, a modyfikacje aktualnych metadanych są dozwolone. Jeszcze innym rozwiązaniem jest zastosowanie trwałej struktury danych do reprezentacji metadanych – struktury, która pomimo modyfikacji zachowuje historię wersji, dzięki czemu można pozwolić na dalsze modyfikacje podczas kopiowania ustalonej wersji [8].

4.1.3. Dopasowanie podziału zasobów przy zmianie liczby rdzeni

SeastarFS nie jest elastyczny na zmiany liczby rdzeni. Podział systemu wykonuje się tylko raz podczas jego tworzenia. Przy zwiększeniu liczby rdzeni system będzie nadal działać poprawnie, jednak nie będzie w stanie poprawnie korzystać z dodatkowych rdzeni.

Dużym problemem jest jednak zmniejszenie liczby rdzeni – powoduje to, że niektóre domeny przestają być obsługiwane i ich dane stają się niedostępne. Używanie systemu plików tylko na części domen może spowodować pojawienie się w katalogu głównym danych niezgodnych z obecnie nieaktywnymi domenami.

Najprostsze rozwiązanie wymagałoby przebudowania struktury systemu podczas startu z niewłaściwą liczbą rdzeniu – zmiany położenia metadanych i danych na dysku zgodnie z nowym podziałem, a także ewentualną zmianą domeny niektórych plików i katalogów, czyli zmiany właściciela tych danych.

4.2. Podziękowania

SeastarFS był wzorowany na podstawie szkicu projektu autorstwa Aviego Kivity'ego. Tworzony był w ścisłej współpracy z Piotrem Sarną. Chcielibyśmy podziękować im za ich wartościowy wkład.

Dodatek A

Organizacja pracy zespołu

W tym dodatku, zostały opisane podział obowiązków, metody organizacji pracy oraz sposób współpracy z klientem.

A.1. Podział obowiązków

1. Michał Niciejewski:
    - część instancji backendu tj. operacje otwierania, zamykania, odczytu i zapisu pliku,
    - część mechanizmu kompakcji tj. realizacja kompakcji na wybranych klastrach,
    - obsługa bootstrap-recordu – odpowiednika superbloku w innych systemach plików,
    - implementacja aplikacji `fs_perf`,
    - testy jednostkowe do powyższych.
2. Krzysztof Małysa:
    - główna część instancji backendu tj. synchronizacja, interfejs, operacje tworzenia i usuwania plików i katalogów, przechodzenie po katalogu,
    - naprawianie systemu plików po awarii,
    - struktura metadanych w pamięci operacyjnej,
    - zapisywanie i odczytywanie metadanych z dysku,
    - testy jednostkowe do powyższych.
3. Aliaksandr Sarokin:
    - cała instancja frontendu tj. podział na rdzenie, interfejs, połączenie z backendem, utworzenie i uruchomienie systemu,
    - zapewnienie ciągłej integracji (*ang.* continuous integration) systemu w repozytorium,
    - stworzenie aplikacji `sfs_io_tester`,
    - przeprowadzenie testów opóźnienia za pomocą aplikacji `sfs_io_tester` i `io_tester` oraz ich analiza,
    - testy jednostkowe.
4. Wojciech Mitros:

- część instancji backendu, w szczególności: alokator klastrów, operacja zmiany rozmiaru pliku,
- część mechanizmu kompakcji, w szczególności: analiza danych na dysku i kontrola uruchamiania kompakcji,
- przeprowadzenie testów wydajnościowych za pomocą aplikacji `fs_perf` i ich analiza,
- testy jednostkowe.

A.2. Metody organizacji pracy

Repozytorium z kodem źródłowym projektu znajduje się na platformie GitHub. Do podziału zadań pomiędzy członkami zespołu, rewizji kodu źródłowego i ciągłej integracji użyto narzędzia oferowane przez platformę GitHub. Synchronizacja wewnątrz zespołu odbywała się dwa razy w tygodniu, a w ostatnim miesiącu codziennie.

A.3. Współpraca z klientem

Na początku powstawania projektu było przeprowadzone kilka spotkań w biurze firmy, gdzie omówiono kwestie organizacyjne. Dalej spotkania przeszły do trybu zdalnego poprzez Google Meet i Google Hangouts z częstotliwością co tydzień. Szybkie konwersacje tekstowe z klientem były prowadzone za pośrednictwem narzędzia Slack. Przegląd kodu źródłowego przez klienta odbywał się przy pomocy GitHub oraz listy mailingowej `seastar-dev`.

Dodatek B

Zawartość płyty CD

Dołączona płyta CD zawiera następujące materiały:

- Kod zrębu Seastar umożliwiający uruchomienie naszego rozwiązania: `seastar/`
- Kod systemu plików:
  - `seastar/include/seastar/fs/` – pliki nagłówkowe zawierające interfejs dla użytkownika biblioteki,
  - `seastar/src/fs/` – implementacja.
- Testy jednostkowe i atrapy obiektów: `seastar/tests/unit/fs_*`
- Aplikacje do testowania efektywności rozwiązań:
  - `seastar/apps/fs_perf/` – aplikacja sprawdzająca efektywność rozwiązania przy długim okresie użytkowania,
  - `seastar/apps/cluster_reading_perf/` – aplikacja testująca efektywność różnych rozwiązań realizacji odczytu fragmentów danych z klastrów (więcej informacji w podrozdziale 2.5.2),
  - `seastar/apps/sfs_io_tester/` – aplikacja `io_tester` zaimplementowana przez ScyllaDB dostosowana do SeastarFS. Mierzy opóźnienie operacji oraz przepustowość,
- Instrukcje:
  - `README.md` – kompilacja i sposób uruchamiania testów i aplikacji,
  - `tutorial.md` – poradnik wprowadzający do pisania aplikacji używających SeastarFS.
- Prezentacje:
  - `prezentacje/postepy_1.pdf` – pierwsza prezentacja z postępami,
  - `prezentacje/postepy_2.pdf` – druga prezentacja z postępami,
  - `prezentacje/koncowa.pdf` – prezentacja końcowa.
- Wizja: `wizja.pdf`
- Praca: `praca_licencjacka.pdf`
- Plakat: `plakat.pdf`

Dodatek C

Testy poprawnościowe i atrapy

W celach testowania systemu implementowaliśmy w czasie trwania projektu testy jednostkowe i atrapy (*ang.* mock objects).

C.1. Testy jednostkowe

- `fs_block_device_test` – testy struktury używanej do czytania danych bezpośrednio z urządzenia blokowego z pominięciem pamięci podręcznej jądra,
- `fs_bootstrap_record_test` – testy operacji na bootstrap recordzie – weryfikacja poprawności odczytywanego i zapisywanego bootstrap recordu,
- `fs_cluster_allocator_test` – testy alokatora klastrów używanego w backend shardzie,
- `fs_filesystem_test` – testy frontendu systemu plików,
- `fs_log_bootstrap_test` – testy procesu bootstrappingu systemu plików,
- `fs_to_disk_buffer_test` – testy struktury używanej do wygodnego dopisywania danych do wskazanego klastra,
- `fs_metadata_to_disk_buffer_test` – testy struktury dziedziczącej po `to_disk_buffer` dostosowanej do wygodnego dodawania wpisów metadanych do klastra logu metadanych,
- `fs_mock_metadata_to_disk_buffer_test` – test atrapy `metadata_to_disk_buffera`,
- `fs_path_test` – testy operacji na ścieżkach plików – sprowadzanie ścieżki do postaci kanonicznej, wyodrębnianie ostatniej części ścieżki (nazwy pliku lub katalogu), itp.
- `fs_truncate_test` – testy operacji `truncate()` i `read()`,
- `fs_write_test` – testy operacji `write()` i `read()`.

C.2. Atrapy

- `fs_mock_metadata_to_disk_buffer` – atrapa `metadata_to_disk_buffer`, między innymi umożliwia przeglądanie listy dodawanych wpisów w logu metadanych po wykonaniu operacji,

- `fs_mock_cluster_writer` – atrapa struktury `cluster_writer` służącej do dodawania danych do logu danych, umożliwia przeglądanie pozycji i zawartości zapisu na dysku wykonywanych operacji średniego zapisu,
- `fs_mock_block_device` – atrapa struktury `block_device`, umożliwia wykonywanie operacji dyskowych, takich jak zapis klastra logu metadanych, czy zapis danych z operacji `write`, w pamięci i łatwy dostęp do tych danych.

Bibliografia

[1] Inc Advanced Micro Devices. *2nd Gen AMD EPYC^{TM} Processors | EPYC^{TM} 7002 Series | AMD.* URL: https://www.amd.com/en/processors/epyc-7002-series (term. wiz. 16. 06. 2020).

[2] Jens Axboe. „Efficient IO with io_uring". W: (15 paź. 2019). URL: https://kernel.dk/io_uring.pdf (term. wiz. 08. 06. 2020).

[3] Jonathan Corbet. „Ringing in a new asynchronous I/O API". W: (15 sty. 2019). URL: https://lwn.net/Articles/776703/ (term. wiz. 08. 06. 2020).

[4] Intel Corporation. *All Intel® Xeon® Processors.* URL: https://www.intel.com/content/www/us/en/products/processors/xeon/view-all.html (term. wiz. 16. 06. 2020).

[5] Glauber Costa. „Designing a Userspace Disk I/O Scheduler for Modern Datastores: the Scylla example (Part 1)". W: (14 kw. 2016). URL: https://www.scylladb.com/2016/04/14/io-scheduler-1/ (term. wiz. 08. 06. 2020).

[6] Glauber Costa. „Designing a Userspace Disk I/O Scheduler for Modern Datastores: the Scylla example (Part 2)". W: (29 kw. 2016). URL: https://www.scylladb.com/2016/04/29/io-scheduler-2/ (term. wiz. 08. 06. 2020).

[7] Glauber Costa. „The Scylla I/O Scheduler – Better Latencies Under Any Circumstance". W: (19 kw. 2018). URL: https://www.scylladb.com/2018/04/19/scylla-i-o-scheduler-3/ (term. wiz. 08. 06. 2020).

[8] J R Driscoll i in. „Making Data Structures Persistent". W: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing.* STOC '86. Berkeley, California, USA: Association for Computing Machinery, 1986, s. 109–121. ISBN: 0897911938. DOI: 10.1145/12130.12142. URL: https://doi.org/10.1145/12130.12142.

[9] Ecma International. *ECMAScript 2015 Language Specification.* Czer. 2015. URL: http://www.ecma-international.org/ecma-262/6.0/index.html (term. wiz. 08. 06. 2020).

[10] Avi Kivity. „Qualifying Filesystems for Seastar and ScyllaDB". W: (9 lut. 2016). URL: https://www.scylladb.com/2016/02/09/qualifying-filesystems/ (term. wiz. 08. 06. 2020).

[11] Stellar Data Recovery. *SSD Vs HDD - Performance comparison of Solid State Drives and Hard Drives.* URL: https://www.stellarinfo.co.in/kb/ssd-vs-hdd.php (term. wiz. 16. 06. 2020).

[12] Mendel Rosenblum i John K. Ousterhout. „The Design and Implementation of a Log-Structured File System”. W: *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*. Red. Henry M. Levy. ACM, 1991, s. 1–15. ISBN: 0-89791-447-3. DOI: 10.1145/121132.121137. URL: https://doi.org/10.1145/121132.121137.

[13] ScyllaDB. *4 Nodes of Scylla on AWS i3.metal vs 40 nodes of Cassandra on i3.4xlarge*. URL: https://www.scylladb.com/product/benchmarks/aws-i3-metal-benchmark/ (term. wiz. 08. 06. 2020).

[14] ScyllaDB. *Compaction*. URL: https://docs.scylladb.com/kb/compaction/ (term. wiz. 08. 06. 2020).

[15] ScyllaDB. *Message Passing*. URL: http://seastar.io/message-passing/ (term. wiz. 11. 06. 2020).

[16] ScyllaDB. *Scylla vs. Cassandra at Samsung: YCSB Benchmark*. URL: https://www.scylladb.com/product/benchmarks/samsung-benchmark/ (term. wiz. 08. 06. 2020).

[17] ScyllaDB. *Shared-nothing Design*. URL: http://seastar.io/shared-nothing/ (term. wiz. 08. 06. 2020).

[18] Abraham Silberschatz, Peter Baer Galvin i Greg Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018. Rozd. 8.5 Deadlock Prevention. ISBN: 978-1-118-06333-0. URL: http://os-book.com/OS10/index.html.

[19] N. Suneja. „Scylladb optimizes database architecture to maximize hardware performance”. W: *IEEE Software* 36.4 (2019), s. 96–100. URL: https://ieeexplore.ieee.org/document/8738153.

[20] Kristian Vättö. *NVMe vs AHCI: Another Win for PCIe*. URL: https://www.anandtech.com/show/7843/testing-sata-express-with-asus/4#:~:text=The%20biggest%20advantage%20of%20NVMe,in%20~2.5%C2%B5s%20of%20additional%20latency (term. wiz. 16. 06. 2020).

[21] Victoria Zhislina. *Why has CPU frequency ceased to grow?* URL: https://software.intel.com/content/www/us/en/develop/blogs/why-has-cpu-frequency-ceased-to-grow.html (term. wiz. 16. 06. 2020).