ZNS를 이용한 키-밸류 스토어 성능 개선 연구

팀 명: System_A

부산대학교 전기컴퓨터공학부 정보컴퓨터공학전공

201724485 배재홍

201224507 이재석

201724588 조준호

지도교수: 안성용

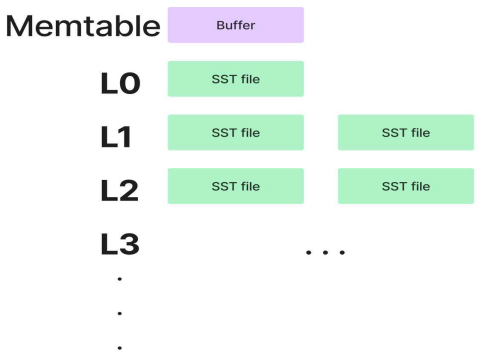
목차

1. 과제 배경 및 목적

1.1 과제 배경

최근 키-밸류 스토어(key-value stores)는 고성능, 유연성 등과 같은 여러 이점이 있어 새로운 기술들에 많이 사용되고 있다. 특히 NoSQL 데이터베이스(DataBase) 기술로 비정형(Unstructured)의 빅데이터(Big Data)를 저장하는데 적합하다. 대표적인 키-밸류 스토어인 RocksDB나 LevelDB는 LSM-Tree(Log-Structured Merge-Tree) 기반 데이터베이스이다. 이러한 데이터베이스는 낸드 플래시 메모리(NAND Flash Memory) 기반 SSD(Solid State Drive)에서 높은 입/출력 성능을 얻을 수 있다.

LSM-Tree는 순차 쓰기(Sequential Write)를 수행하여 데이터(Data)를 write 할 때 높은 성능을 보인다. 그리고 레벨(Level) 별로 데이터를 저장하는데, 각 레벨은 이전 레벨보다 더 많은 데이터를 저장할 수 있다. 기본적으로 이전 레벨의 10배에 해당하는 크기만큼 데이터를 더 많이 저장할 수 있다.

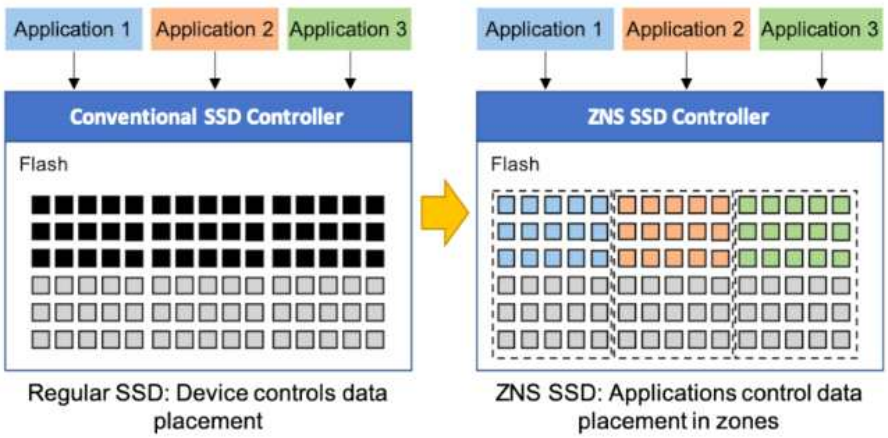


[그림-1] RocksDB의 LSM-Tree 구조

[그림-1]은 RocksDB에서 사용하는 LSM-Tree의 구조를 그림으로 나타낸 것이다. Memtable은 메모리(Memory) 내부에 존재하며 write 발생 시 키-밸류들을 저장하는 버퍼(Buffer) 역할을 한다. L0, L1 등은 각 레벨을 나타낸 것이고 SST file은 RocksDB에서 사용하는 키-밸류들이 키 순서에 따라 저장된 파일이다. 우선 데이터 write가 발생하면 Memtable에 우선적으로 write 되며 Memtable의 버퍼가 가득 찰 시 SST file로 플러시(Flush) 된다. 그리고 각 레벨이 SST file로 가득 차게 되면 바로 다음 레벨과 컴펙션(Compaction)을 수행하게 된다. Compaction은 SST file을 아래 레벨로 옮기는 과정이다. Compaction 수행 시 옮기게 될 SST file과 바로 아래 레벨 SST file들 중 키

범위가 겹치는 모든 SST file이 병합(Merge) 되어 새로운 SST file들이 생성된다. 예를 들어 L1과 L2 사이에 compaction이 일어난다고 가정하면, L1에서 compaction할 SST file이 선택되고 그 SST file과 키 범위가 겹치는 모든 L2의 SST file들이 선택된다. 그리고 L1, L2에서 선택된 SST file들이 병합돼 새로운 SST file들이 생성되고 L2에 저장된다.

SSD는 HDD(Hard Disk Drive)와 달리 덮어쓰기(Overwrite)가 불가능하여 데이터 수정 시 해당 데이터를 무효화(Invalid) 시키고 새로운 곳에 데이터를 write 해야 하는 특성이 있다. 그리고 invalid된 영역은 erase 후 다시 write 할 수 있는데 이 erase 단위가 write 단위보다 훨씬 크다. 그래서 erase 할 때 erase 범위 내 유효(Valid) 한 데이터들은 다른 곳에 복사(copy) 한 후 erase를 수행해야 한다. 이러한 과정을 Garbage Collection(이하 GC)라 한다. 이렇게 의도하지 않은 write가 발생하는 현상을 쓰기증폭(Write-Amplification)이라고 하며 copy로 인한 성능 overhead가 굉장히 크다 . 이러한 overhead를 줄이기 위해 GC는 SSD의 남은 용량이 일정수준 이하일 때 수행되고 invalid 데이터가 많은 영역을 우선적으로 수행한다.



[그림-2] (좌)기존 SSD와 (우)ZNS SSD 저장방식 비교

[그림-2]는 기존 SSD와 ZNS(Zoned NameSpace) SSD의 저장 방식 차이를 보여주는 그림이다. 기존 SSD는 여러 애플리케이션(Application)이 데이터를 write 할 때 순서에 상관없이 Flash에 write 하는 것을 볼 수 있다. 반면 ZNS SSD는 flash를 zone이라는 논리적(Logical) 영역으로 나누어 각 애플리케이션의 데이터가 분할된 zone에 write 되는 것을 볼 수 있다. 이렇게 한 zone에 한 애플리케이션의 데이터만 write 되면 해당 zone의 데이터들은 비슷한 수명(lifetime)을 갖게 된다. 그러면 GC 시 overhead를 줄일 수 있는 이점이 있다.

1.2 현 RocksDB 성능

1.2.1 실험환경

본 실험은 ZNS SSD를 에뮬레이트(Emulate) 할 수 있는 가상머신(Virtual Machine)에서 진행됐다. 가상머신은 FEMU라는 프로그램을 사용했고 메모리에 지연(latency)을 주는 방식으로 ZNS SSD를 에뮬레이트 한다.

CPU 코어 수	32
Memory 크기	32GB
저장소(Storage)	에뮬레이트 된 ZNS SSD
커널(Kernel) 버전(Version)	Linux Kernel 5.10.179

[표-1] 가상머신 사양

용량(Capacity)	16GB
섹터(Sector) 크기	512Bytes
Zone 크기	128MB
Zone 개수	128

[표-2] 에뮬레이트 한 ZNS SSD 사양

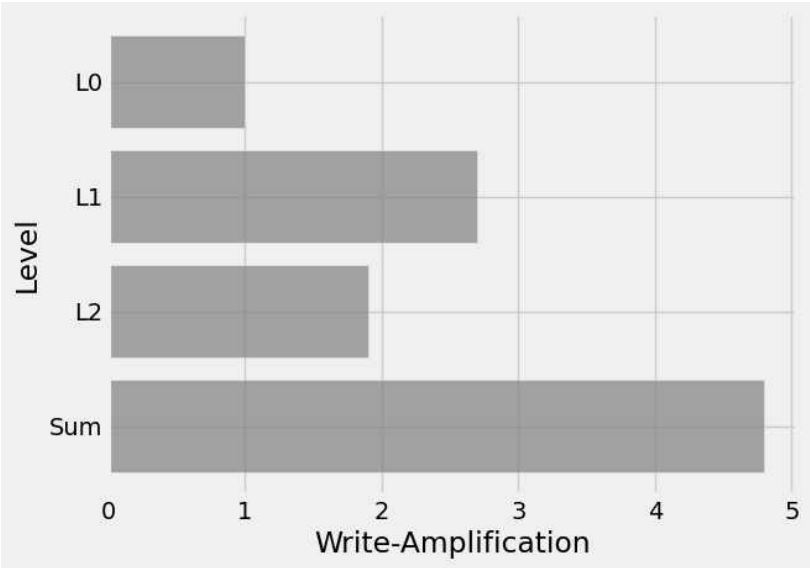
[표-1]은 가상머신의 사양(Specification)을 작성한 표이고 **[표-2]**는 FEMU를 통해 에뮬레이트 한 ZNS SSD의 사양을 작성한 표이다. RocksDB의 성능 평가 툴(Tool)은 RocksDB에 내장된 db_bench를 사용했다.

1.2.2 벤치마크(Benchmark)

키(key) 크기	16Bytes
밸류(value) 크기	100Bytes
키-밸류 개수	100,000,000
SST file 크기	64MB
Level-1 크기	256MB

[표-3] db_bench 옵션(Option)

[표-3]은 db_bench를 사용하여 벤치마크할 때 사용한 옵션이다. 이외에 키-밸류 저장 방식은 비동기(Asynchronous) 랜덤 오더(random order)로 키값을 무작위 순서로 write 했다. 또, 레벨 당 저장 가능한 크기는 10배씩 증가한다.



[그림-3] 벤치마크 중 측정된 로그(Log)

[그림-3]는 벤치마크 중 기록된 로그의 일부분을 그래프로 나타낸 것이다. L0, L1, L2는 레벨을 나타내고 그래프 수치는 각 레벨에서 일어난 Write-Amplification을 나타낸다. Sum은 모든 레벨에서 일어난 Write-Amplification의 총합이다. L0는 1.0으로 Write-Amplification이 발생하지 않았고 L1, L2는 각각 2.7, 1.9배 만큼 Write가 더 발생했다. 시간이 지나 write가 발생할수록 Write-Amplification도 증가하게 될 것이며 이는 overhead의 원인이 된다.

1.3 과제 목표

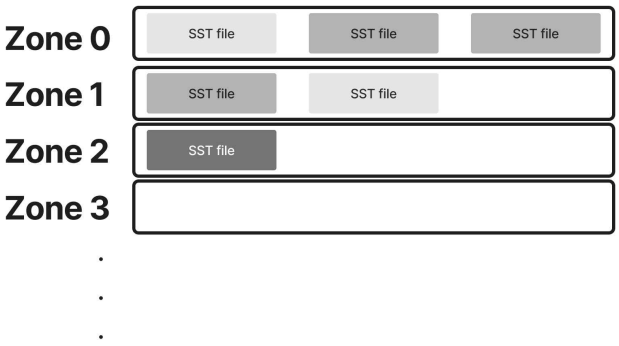
SSD는 플래시 메모리 기반 저장 장치로 write 횟수가 정해져있어 수명이 HDD에 비해 짧다. 그리고 GC 시 copy에 의한 성능 overhead가 발생한다. 이러한 단점을 극복하기 위해 ZNS SSD가 개발됐다. 그리고 현재도 GC overhead를 줄이기 위한 연구, 새로운 데이터가 write 될 때 zone 할당 정책 연구 등 SSD의 수명과 성능을 개선하기 위한 여러 연구가 활발히 진행되고 있다.

현 RocksDB는 ZNS SSD를 사용할 때 ZenFS라는 내장된 파일 시스템(File System)을 사용한다. ZenFS에도 zone 할당 정책 및 GC 시 희생(victim) zone 선택 정책 등이 구현되어 있다. 하지만 현재 구현된 방식도 **[그림-3]**에서 볼 수 있듯이 Write-Amplification에 의한 overhead가 발생한다. SSD 특성상 이러한 Write-Amplification으로 인한 overhead를 완전히 피할 수 없다. 하지만 본 과제를 통해 Write-Amplification을 줄여 성능 향상을 하고자 한다.

2. RocksDB 코드분석

RocksDB를 통하여 ZNS SSD를 사용할 때 성능을 개선하기 위해선 현재 RocksDB에서 사용하고 있는 정책이 무엇인지 알아야 한다. 따라서 RocksDB의 코드(Code)를 분석한다. 우선적으로 RocksDB에 내장된 ZenFS의 코드를 통해 데이터 write 시 zone 할당 정책, GC 과정 및 GC 시 victim zone 선정 기준에 대해 분석했다.

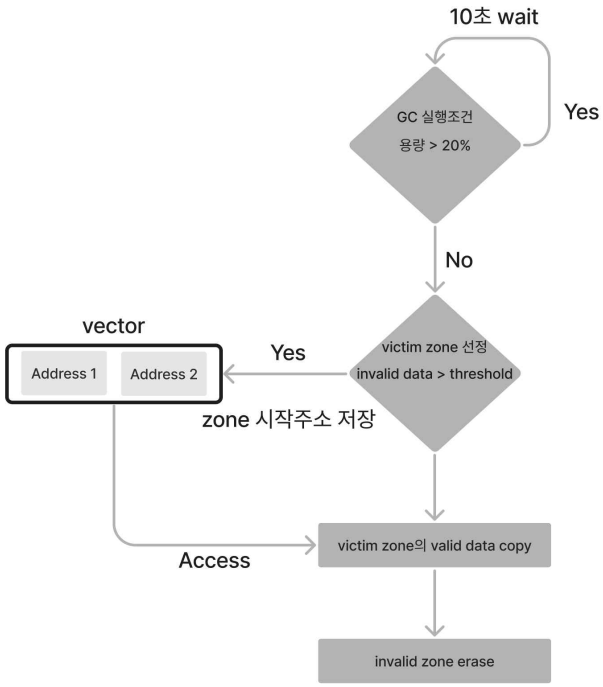
2.1 ZenFS의 zone 할당 정책



[그림-4] 현 ZenFS의 zone 할당 정책

[그림-4]는 현재 ZenFS에 구현돼있는 zone 할당 정책을 나타낸 것이다. 각 zone에는 SST file이 최대 3개 들어갈 수 있다고 가정한다. 그리고 같은 색을 가진 SST file은 같은 수명(lifetime)을 갖는다. SST file의 수명은 레벨에 따라 다르게 측정된다. 낮은 레벨은 높은 레벨보다 사이즈가 작아 compaction이 자주 일어난다. 그러면 낮은 레벨의 SST file은 자주 삭제되므로 비교적 짧은 수명을 갖게 된다. 현 ZenFS에서는 SST file을 write 할 때 GetBestOpenZoneMatch() 함수를 사용해 모든 zone을 스캔(Scan) 하여 가장 수명이 비슷한 SST file이 많은 zone에 write 한다. 이러한 방식은 수명이 짧은 SST file이 모여있는 zone을 형성할 것이다. 그리고 수명이 짧은 SST file들은 invalid 될 확률이 높아 해당 zone에 대해 GC시 valid 데이터 copy를 줄여 overhead를 줄일 수 있다.

2.2 ZenFS의 GC 과정



[그림-5] 현 ZenFS의 GC 과정 Flow Chart

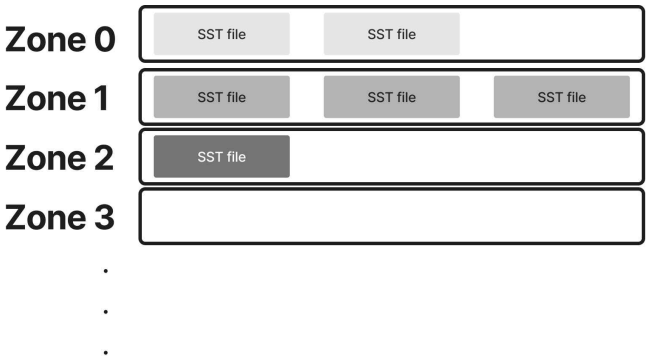
[그림-5]는 현재 ZenFS에 구현돼있는 GC 과정을 Flow Chart로 나타낸 것이다. 먼저 ZNS SSD의 남은 용량이 20% 미만일 때 GC가 발생하게 된다. 남은 용량이 20% 미만이라면 GC 대상이 될 victim zone을 찾는다. 이는 모든 zone을 스캔하여 zone의 invalid 데이터 비율이 임계값(threshold)보다 큰 zone을 victim zone으로 선정해 zone의 시작 주소를 벡터(vector)에 저장한다. 다음 victim zone 선정이 완료되면 MigrateFileExtents() 함수를 호출해 victim zone에 존재하는 valid data를 다른 zone에 copy 하고 해당 zone을 invalid 시킨다. 이때 **2.1 ZenFS의 zone 할당 정책**에 따라 가장 수명이 비슷한 zone에 copy 하게 된다. copy가 완료되면 ResetUnusedIOZone() 함수를 호출해 invalid zone을 모두 erase 한다.

2.3 GC threshold

현재 ZenFS는 각 zone의 invalid 데이터 비율이 threshold보다 큰 zone을 victim zone으로 선정한다. 이 threshold 값은 간단한 수식으로 나타나는데, $100-3*(20-x)$이다. 이때 x는 zone의 남은 용량이다.

3. 과제 세부 목표

3.1 Zone 할당 정책 개선



[그림-6] 레벨별 zone 할당 정책

앞서 보았듯이 현 ZenFS에서는 수명이 가장 비슷한 데이터들을 한 zone에 저장하는 정책을 사용 중이다. 이를 개선해 [그림-6]과 같이 레벨마다 zone을 따로 할당하여 각 zone에 같은 수명을 가진 데이터만 저장하는 정책을 구현할 계획이다. 이러한 정책을 통해 GC 시 zone의 valid 데이터를 최소화하여 copy overhead를 줄이는 효과를 기대할 수 있다.

3.2 Victim zone 선정 기준 개선

현재 ZenFS는 victim zone 선정 기준으로 간단한 일차식의 threshold를 사용한다. 이 식을 고도화시켜 victim zone 선정 시 valid 데이터 copy를 줄이는 방향으로 개선하고자 한다.

4. 개발 일정 및 역할 분담

4.1 개발 일정

<table>
<tr><th colspan="5">6월</th><th colspan="4">7월</th><th colspan="5">8월</th><th colspan="4">9월</th></tr>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>1</td><td>2</td><td>3</td><td>4</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td colspan="5" rowspan="2">RocksDB 코드 분석 및 관련 논문 공부</td><td colspan="4" rowspan="2"></td><td colspan="5" rowspan="2"></td><td colspan="4" rowspan="2"></td></tr>
<tr></tr>
<tr><td colspan="5" rowspan="2"></td><td colspan="3" rowspan="2">Zone 할당 정책 구현</td><td colspan="2" rowspan="2"></td><td colspan="5" rowspan="2"></td><td colspan="4" rowspan="2"></td></tr>
<tr></tr>
<tr><td colspan="5" rowspan="2"></td><td colspan="3" rowspan="2"></td><td colspan="2" rowspan="2">중간 보고서 작성</td><td colspan="5" rowspan="2"></td><td colspan="4" rowspan="2"></td></tr>
<tr></tr>
<tr><td colspan="5" rowspan="2"></td><td colspan="4" rowspan="2"></td><td colspan="4" rowspan="2">GC 개선방안 연구 및 구현</td><td rowspan="2"></td><td colspan="3" rowspan="2"></td><td colspan="2" rowspan="2"></td></tr>
<tr></tr>
<tr><td colspan="5" rowspan="2"></td><td colspan="4" rowspan="2"></td><td colspan="4" rowspan="2"></td><td colspan="3" rowspan="2">구현 코드 테스트 및 수정</td><td colspan="2" rowspan="2"></td></tr>
<tr></tr>
<tr><td colspan="5" rowspan="2"></td><td colspan="4" rowspan="2"></td><td colspan="4" rowspan="2"></td><td rowspan="2"></td><td colspan="2" rowspan="2">성능 측정</td><td colspan="2" rowspan="2"></td></tr>
<tr></tr>
<tr><td colspan="5" rowspan="2"></td><td colspan="4" rowspan="2"></td><td colspan="4" rowspan="2"></td><td rowspan="2"></td><td colspan="3" rowspan="2">최종 보고서 작성 및 발표 준비</td></tr>
<tr></tr>
</table>

4.2 역할 분담

배재홍	- RocksDB 코드 분석 및 동작분석 - GC 최적화 구현 - 성능 평가 및 결과 분석
이재석	- RocksDB 코드 분석 및 동작분석 - GC threshold 계산 식 설계 - 테스트 및 디버깅
조준호	- ZenFS 코드 분석 및 동작분석 - Zone 할당 정책 구현 - GC 시작 수준 관련 논문 공부

[참고 자료]

[1] [rocksDB] rocksDB의 소개와 구조
(https://headf1rst.github.io/rocksdb/rocksDB_compaction_spaceAmplification/)

[2] Facebook의 Key-Value DB, "RocksDB" (https://hyj3463.tistory.com/26)

[3] RocksDB Wiki (https://github.com/facebook/rocksdb/wiki)

[4] Zoned Storage Document (https://zonedstorage.io/docs/introduction)

[5] FEMU 참고자료 (https://github.com/vtess/FEMU)

[6] 벤치마크 참고자료
(https://github.com/EighteenZi/rocksdb_wiki/blob/master/Benchmarking-tools.md)

[7] Sungyoung Ahn 등, "Efficient Key-Value Data Placement for ZNS SSD", Appl, Sci, 2021.

[8] Youngjae Kim 등, "Compaction-Aware Zone Allocation for LSM based Key-Value Store on ZNS SSDs", HotStorage, 2022.

[9] 신동군 등, "Zoned Namespace SSD를 위한 LSM-tree 최적화 기법",
한국소프트웨어종합학술대회, 2021.