ZNS를 이용한 키-밸류 스토어 성능 개선 연구

분과: D
팀 명: System_A

부산대학교 전기컴퓨터공학부 정보컴퓨터공학전공

201724485 배재홍

201224507 이재석

201724588 조준호

지도교수: 안성용

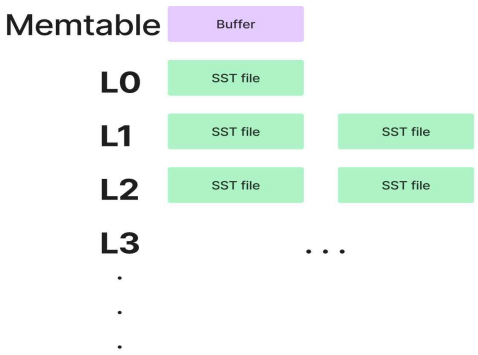
목차

1. 요구조건 및 제약 사항 분석에 대한 수정사항

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.

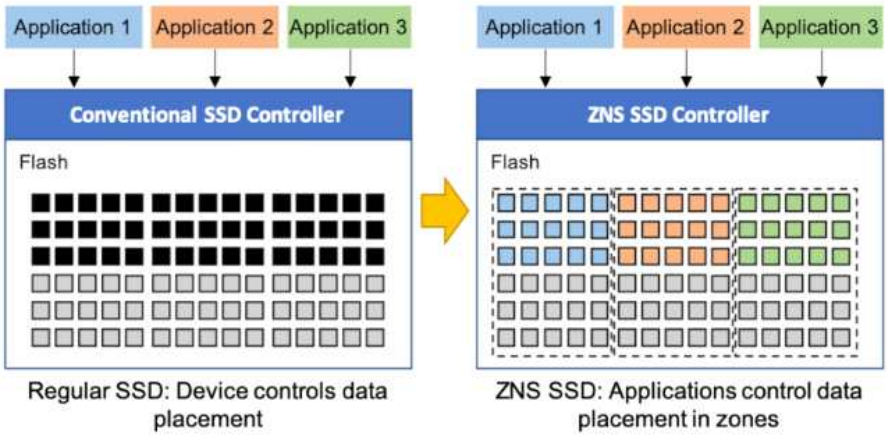


[그림-1] RocksDB의 LSM-Tree 구조

[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.



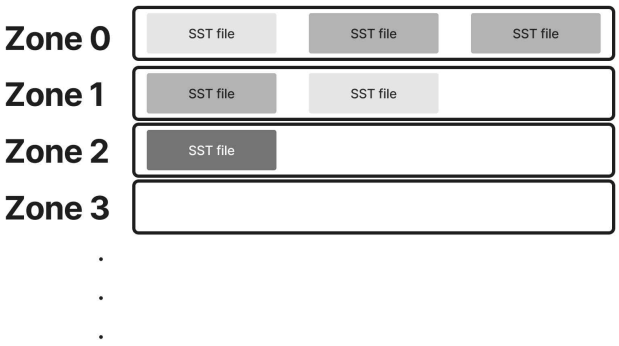
[그림-2] (좌)기존 SSD와 (우)ZNS SSD 저장방식 비교

[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 과제 목표

SSD는 플래시 메모리 기반 저장 장치로 write 횟수가 정해져있어 수명이 HDD에 비해 짧다. 그리고 Garbage Collection 시 copy에 의한 성능 overhead가 발생한다. 이

러한 단점을 극복하기 위해 ZNS SSD가 개발됐다. 그리고 현재도 Garbage Collection overhead를 줄이기 위한 연구, 새로운 데이터가 write 될 때 Zone 할당 정책 연구 등 SSD의 수명과 성능을 개선하기 위한 여러 연구가 활발히 진행되고 있다.



[그림-3] 현 ZenFS의 zone 할당 정책

현 RocksDB는 ZNS SSD를 사용할 때 ZenFS라는 내장된 파일 시스템(File System)을 사용한다. 현 ZenFS는 **[그림-3]과** 같이 같은 Zone에 유사한 LifeTime의 SST File을 저장해 Garbage Collection 시 Migration을 최소화하고자 한다. 이를 개선하기 위해 새로운 여러 Zone 할당 정책 들을 구현 및 테스트해보는 것이 본 과제의 목표이다.

1.3 제약 사항 및 대책

현재 RocksDB는 다양한 read/write 환경에 대응하기 위해 많은 옵션들을 제공한다. 하지만 제공하는 옵션이 너무 많아 모든 옵션들에 대해 성능 평가가 불가능하다. 따라서 본 과제는 가장 기본적인 read/write 옵션에 대해서 성능 평가를 진행한다.

2. 설계 상세화 및 변경 내역

2.1 Zone 할당 정책에 따른 성능평가

기존의 과제 목표 중 하나는 zone 할당 정책 개선을 통한 write amplification을 줄이는 것이었다. 해당 목표를 달성하기 위해 기존 Zone 할당 정책 이외의 할당 정책 두 가지를 구현하고 성능 평가를 진행했다.

2.2 실험 환경

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)	32GB
섹터(Sector) 크기	512Bytes
Zone 크기	128MB
Zone 개수	256

[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.

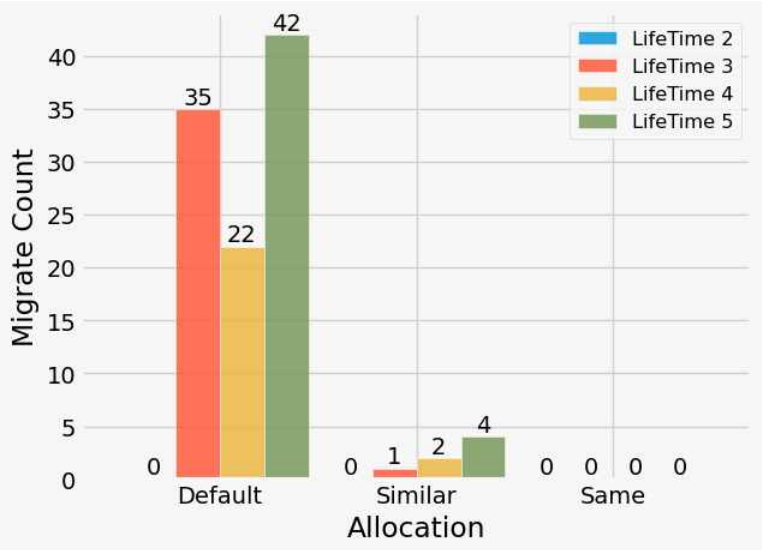
2.3 벤치마크

키(key) 크기	16Bytes
밸류(value) 크기	100Bytes
키-밸류 개수	320,000,000
SST file 크기	64MB
Level-1 크기	256MB

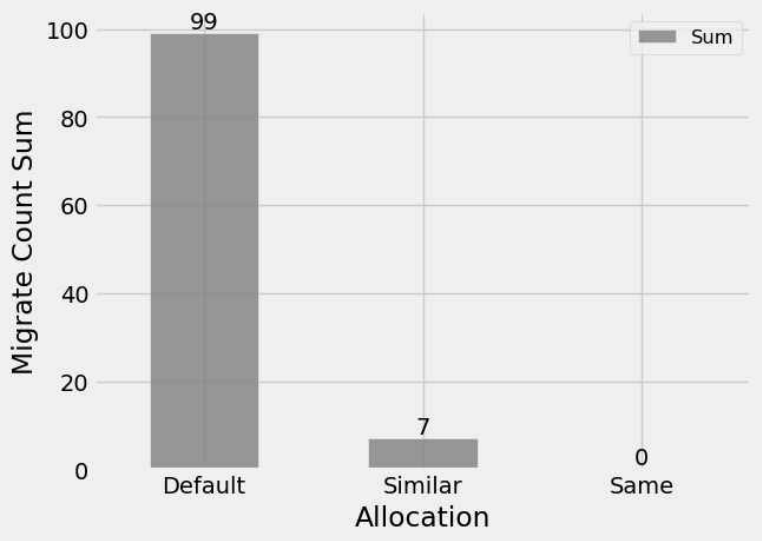
[표-3] db_bench 옵션(Option)

[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 2배씩 증가한다.

2.4 실험 결과



[그래프-1] Zone 할당 정책에 따른 Migration된 SST File의 LifeTime



[그래프-2] Zone 할당 정책에 따른 Migration된 SST File 수

[그래프-1]과 [그래프-2]는 Zone 할당 정책에 따른 Migration된 SST File에 대한 실험 결과이다. Default는 현재 ZenFS에서 사용 중인 기본 Zone 할당 정책이고 Similar와 Same은 본 과제 수행 중 구현한 Zone 할당 정책이다. [그래프-1]은 Migration된 SST File을 LifeTime 기준으로 나눠 합산한 그래프이고 [그래프-2]는 Migration된 SST File의 총합을 나타낸 그래프이다.

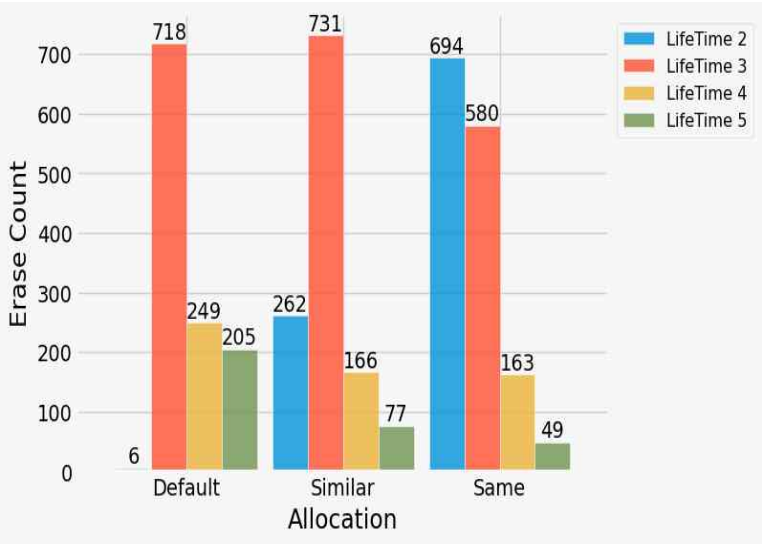
Default	Zone LifeTime 〉 SST File LifeTime
Similar	Zone LifeTime 〉= SST File LifeTime
Same	Zone LifeTime = SST File LifeTime

[표-4] 세 Zone 할당 정책

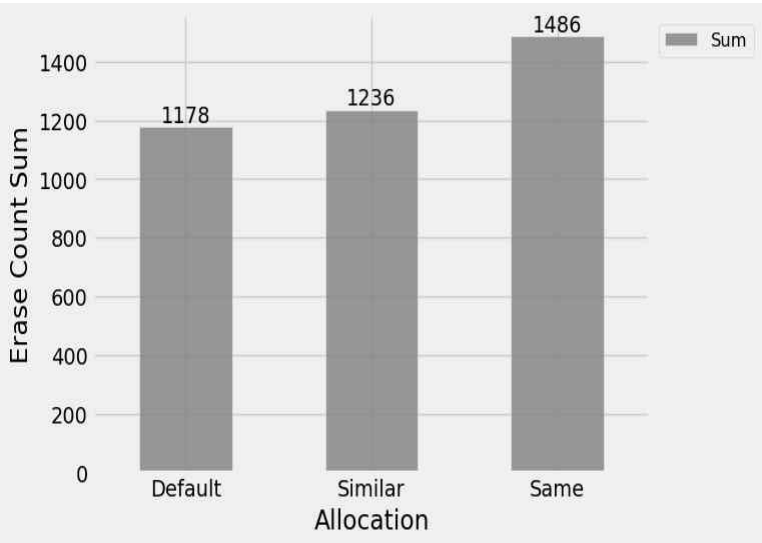
우선 세 Zone 할당 정책의 공통점은 빈 Zone에 처음 write한 SST File의 LifeTime이 Zone의 LifeTime이 된다는 것이다. [표-4]는 세 Zone 할당 정책의 차이를 보여준다. Default는 위 조건을 만족하는 Zone들 중 LifeTime차이가 가장 작은 Zone에 SST File을 write한다. Similar는 LifeTime이 같은 Zone에 우선적으로 write하고 그러한 Zone이 없는 경우 Default와 같이 조건을 만족하는 Zone들 중 LifeTime차이가 가장 작은 Zone에 SST File을 write한다. 마지막으로 Same은 LifeTime이 같은 Zone에만 SST File을 write한다. 세 Zone 할당 정책 모두 위 조건을 만족하는 Zone이 없는 경우 새로운 빈 Zone을 할당한다.

해당 실험 결과로 새로 구현한 Zone 할당 정책에서 Migration이 확연히 줄어든 것을 볼 수 있었다. Migration이 줄어든다면 write amplification도 줄어들게 될 것이다. 하지만 본 실험 중 Zone의 Erase가 Garbage Collection뿐만 아니라 SST File을 삭제할

때도 발생한다는 것을 알아냈다. 따라서 Zone 할당 정책에 따른 Zone Erase 횟수에 대한 실험도 추가적으로 진행했다.



[그래프-3] Zone 할당 정책에 따른 Erase 된 Zone의 LifeTime



[그래프-4] Zone 할당 정책에 따른 Erase 된 Zone 수

위 실험은 SST File이 삭제됨에 따라 모두 invalid 데이터인 Zone을 Erase하는 횟수를 측정한 것이다. **[그래프-3]**은 Zone의 LifeTime 별로 개수를 측정한 것이고 **[그래프-4]**는 Erase가 발생한 Zone의 총합을 나타낸 것이다.

실험 결과 새로 구현한 두 Zone 할당 방식에서 Erase가 더 많이 발생한 것을 알 수 있다. 이는 ZenFS에서 SST File이 Delete될 때마다 Invalid 데이터만 존재하는 Zone을 Erase시키기 때문이다. Similar와 Same은 LifeTime이 같은 Zone에 우선적으로 할당하

므로 Zone에 존재하는 SST File들이 비슷한 시기에 Delete될 가능성이 높다. 따라서 Invalid 데이터만 존재하는 Zone이 생성될 가능성이 증가하고 이에 따라 Erase 수가 증가하게 된다.

2.5 설계 상세화

위 두 실험을 통해 새로운 Zone 할당 정책이 Migration 발생에 의한 write amplification을 줄여주지만 Zone을 Erase하는 횟수가 증가함을 알 수 있었다. 따라서 write amplification과 Erase count의 trade off를 고려해 최적의 성능을 낼 수 있는 Zone 할당 정책을 구현할 것이다.

3. 갱신된 과제 추진 계획

<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="7" rowspan="2">RocksDB 코드 분석 및 관련 논문 공부</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td colspan="4" rowspan="2">Zone 할당 정책 구현</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="5" rowspan="2">Zone 할당 정책 변경에 따른 성능 측정</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="3" rowspan="2">중간보고서 작성</td><td colspan="4" rowspan="2">추가적인 성능 개선점 확인</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="3" rowspan="2">성능 개선</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2" rowspan="2">성능 측정 및 테스트</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2" rowspan="2">최종 보고서 작성 및 발표 준비</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

4. 구성원별 진척도

배재흥	- RocksDB 코드 분석 및 동작분석 - Zone 할당 정책 테스트 - 성능 개선점 구현
이재석	- RocksDB 코드 분석 및 동작분석 - 성능 개선점 확인 - 테스트 및 디버깅
조준호	- ZenFS 코드 분석 및 동작분석 - Zone 할당 정책 구현 - 성능 평가 및 결과 분석

공통	- 보고서 작성 - 발표 준비

5. 보고 시점까지의 과제 수행 내용 및 중간 결과

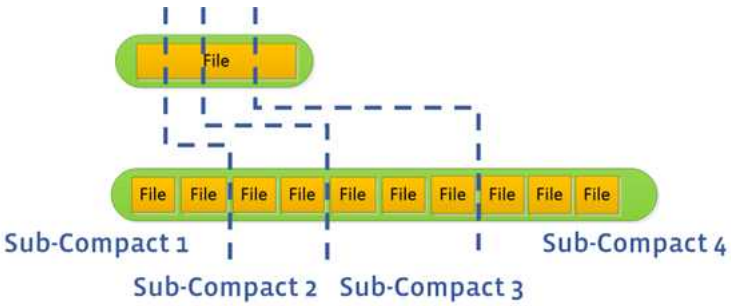
본 과제를 수행하기 위해 write 발생 시 RocksDB와 ZenFS에서 수행되는 함수들을 분석했다. write 발생 시 ZNS SSD까지 write되는 과정은 RocksDB::Write - Flush - Compaction. ZenFS::Append - Invalid - Erase - Garbage Collection이다. 추가적으로 ZenFS의 현재 Zone 할당 정책과 관련된 함수도 분석했다.

5.1 RocksDB Write

RocksDB는 write 성능을 높이기 위해 여러 write 작업을 하나의 묶음으로 처리한다. 이를 위해 key-value가 들어오게 되면 WriteBatch에 쌓이게 된다. 그리고 일정 크기 이상 key-value들이 쌓이면 메모리(MemTable)로 write가 발생하게 된다. 그리고 RocksDB는 예기치 못한 상황에 대비해 데이터 일관성을 보장하기 위해 Log 파일을 기록한다. 이를 Write Ahead Log(이하 WAL)이라 한다. 따라서 write 시 WAL과 SST File 두 개가 모두 write된다. 이 두 write는 순차적으로 발생하며 병렬적으로 실행 가능하다. 따라서 RocksDB는 PipelinedWrite를 사용하여 두 write를 병렬적으로 수행해 보다 빠르게 write한다.

5.2 RocksDB Flush, Compaction

RocksDB의 Flush와 Compaction은 유사한 동작을 수행한다. write 시 key-value들이 MemTable에 write된다. 이 key-value들이 쌓여 MemTable이 가득 차게 되면 Flush가 발생해 SST File이 만들어져 Level 0로 Flush된다. 그리고 각 Level에 SST File이 쌓여 Level의 최대 크기를 넘어가면 바로 다음 level의 SST File과 Compaction이 발생해 다음 Level의 SST File로 합쳐진다. Compaction은 key 범위가 겹치는 SST File들을 합쳐주는 작업이다. 만약 다음 Level에 key 범위가 겹치는 SST File이 없다면 Trivial Move가 발생해 해당 SST File을 Compaction 작업 없이 다음 Level로 넘겨준다.

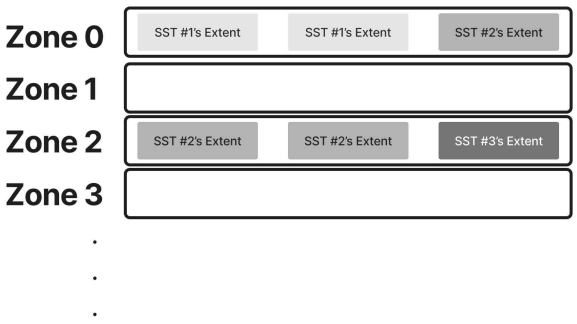


[그림-4] Sub-Compaction

key 범위가 겹치는 SST File이 존재하면 Compaction이 수행된다. Compaction은 크

게 Prepare() - Run() - Install()의 순서로 수행된다. Prepare()는 Sub-Compaction을 위해 key값을 범위로 분할하는 단계이다. Sub-Compaction은 **[그림-4]**에서 볼 수 있듯이 Compaction할 SST File을 분할해 Compaction하는 것이다. Sub-Compaction을 사용하면 한 개의 SST File에 대한 한 Compaction작업을 더 작은 단위의 여러 Compaction으로 나누어 병렬처리하므로 Compaction 속도를 높일 수 있다. Run()은 실제 Compaction을 수행해 결과로 나온 SST File들을 반환해 준다. 마지막 Install()은 각 Level에 Compaction이 발생한 SST File을 제거하고 Run()을 통해 반환된 SST File들을 반영해 준다.

5.3 ZenFS Write



[그림-5] ZenFS SST File 저장 방식

MemTable의 buffer가 가득 차 Level 0로 Flush되면 SST File이 실제 ZNS SSD에 write된다. 이때 SST File은 Extent라는 단위로 나누어져 write된다. **[그림-5]**는 SST File이 Extent단위로 나누어져 실제 ZNS SSD에 저장된 모습의 예시이다. ZenFS는 SST File을 저장할 때 저장할 Zone을 할당한다. 그리고 해당 Zone에 Extent들을 write한다. 이때 할당된 Zone의 용량이 없으면 새로운 Zone을 할당해 남은 Extent들을 write한다. 이러한 방식을 거치면 **[그림-5]**에서 볼 수 있듯이 한 Zone에 여러 SST File의 Extent들이 존재하기도 하고 한 SST File의 Extent들이 여러 Zone에 나누어져 존재하기도 한다.

그리고 **5.1 RocksDB Write**에서 설명한 WAL은 데이터 일관성을 유지하기 위한 데이터이므로 ZNS SSD에 write된다. WAL은 Log 파일로 기록되며 해당 데이터가 무사히 MemTable(메모리)에서 Level 0(ZNS SSD)로 Flush 되면 삭제된다.

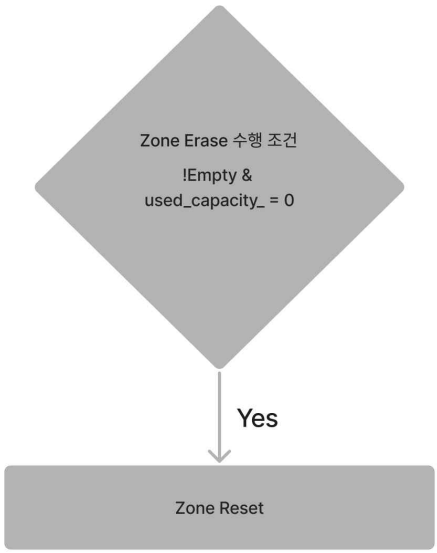
SST File의 write는 BufferedAppend() 함수를 통해 write되고 Log 파일은 SparseAppend() 함수를 통해 write된다. 두 함수는 SparseAppend를 위한 헤더가 존재

하는 것 이외의 기본적인 동작은 동일하다.

5.4 ZenFS Invalid, Erase

HDD와 달리 SSD는 Overwrite가 불가능하다. 따라서 write된 데이터를 수정 또는 삭제하는 경우 해당 데이터를 invalid 하는 과정이 필요하다. 그리고 invalid된 부분을 다시 사용하기 위해 Erase가 필요하다. 이때 Erase 단위가 크기 때문에 invalid 데이터가 쌓이면 한 번에 Erase 한다.

ZenFS에서 데이터를 invalid하는 과정은 간단하다. 해당 데이터가 저장된 Zone의 used_capacity_를 데이터 크기만큼 감소시키고 데이터를 소멸자를 통해 삭제한다. used_capacity_는 현재 Zone의 valid 데이터가 저장된 용량이다. 따라서 Compaction 발생 시 대상이 된 SST File들이 invalid 및 삭제되고 새로운 SST File을 write한다.

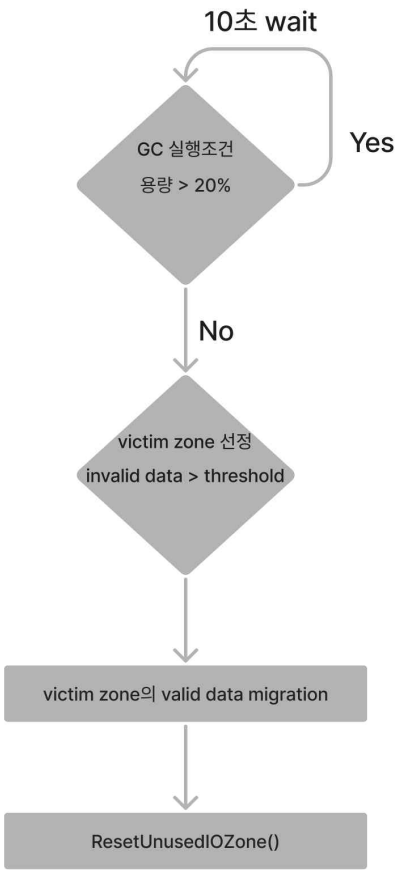


[그림-6] ResetUnusedIOZone() 동작 방식

ZNS SSD에서는 Zone 단위로 Erase가 발생한다. ZenFS는 이러한 특성을 이용하기 위해 각 Zone당 세 변수 start_, wp_, used_capacity_를 저장한다. start_는 Zone의 시작 주소, wp_는 현재 Zone의 write point이고 used_capacity_는 위와 같이 valid 데이터들의 총용량이다. ZenFS에서 Erase는 ResetUnusedIOZone() 함수를 통해 수행된다. ResetUnusedIOZone()의 동작 방식은 **[그림-6]과** 같다. Zone을 검사해 비어있지 않고 used_capacity_=0인, 즉 valid 데이터가 없는 Zone을 Erase(Reset)시킨다. Zone의 start_ = wp_이면 해당 Zone은 비어있는 상태이다. 해당 함수는 SST File이 삭제될 때

와 Garbage Collection 발생 시 수행된다.

5.5 ZenFS Garbage Collection

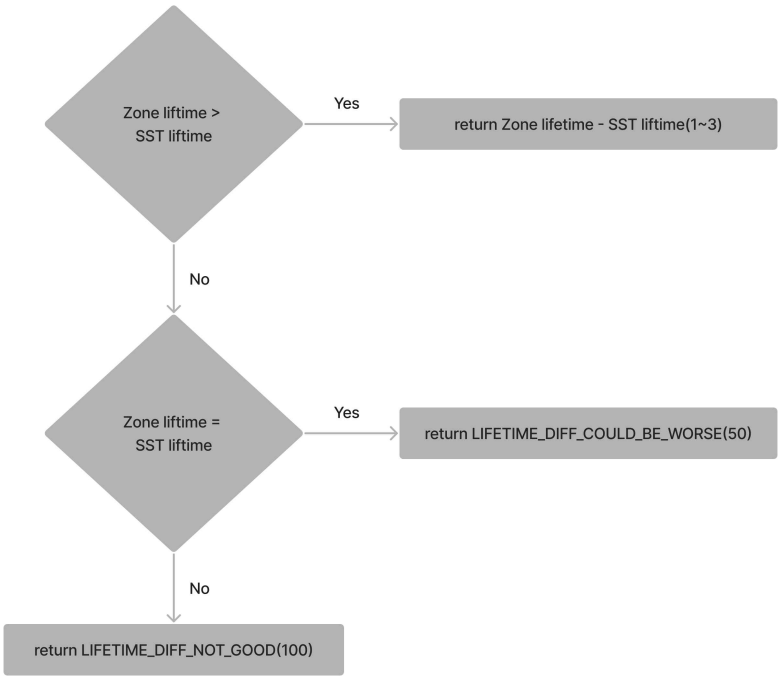


[그림-7] ZenFS Garbage Collection 과정

[그림-7]은 ZenFS의 Garbage Collection 수행 과정을 나타낸 것이다. 먼저 현재 ZNS SSD의 남은 용량이 20% 미만일 때 수행된다. 그리고 모든 Zone을 검사해 invalid 데이터 비율이 임계값보다 큰 Zone을 Victim Zone으로 선정한다. 그리고 해당 Zone에 존재하는 valid 데이터를 다른 Zone에 Migrate하고 ResetUnusedIOZone() 함수를 통해 해당 Zone을 Erase한다.

5.6 ZenFS Zone 할당 정책

[그림-8] (좌)Append 시 (우)Migration 시 Zone 할당 과정



[그림-9] GetBestZoneMatch() - GetLifeTimeDiff() 과정

ZenFS에서 Append와 Migration 시 새로운 Zone 할당이 일어난다. **[그림-9]**에서 볼 수 있듯이 해당 과정은 유사하지만 차이가 있다. 우선 두 과정 모두 GetBestZoneMatch()와 GetLifeTimeDiff()를 호출한다. **[그림-9]**에서 볼 수 있듯이 두

함수는 비어있지 않은 모든 Zone을 검사해 Zone의 lifetime과 새로 write할 SST File의 lifetime을 비교해 가장 최적의 Zone과 lifetime 차이를 반환해 준다. write할 SST File의 lifetime보다 큰 lifetime의 Zone이 있다면 그 Zone들 중 lifetime이 가장 적게 차이나는 Zone이 최적의 Zone이 되고 해당 Zone과 lifetime의 차(1~3)가 반환된다. 그러한 Zone이 없고 lifetime이 같은 Zone이 존재한다면 해당 Zone이 최적의 Zone이 되고 해당 Zone과 50이 반환된다. lifetime이 같은 Zone마저 없고 SST File의 lifetime보다 작은 lifetime의 Zone만 있다면 그러한 마지막으로 비교된 Zone과 100을 반환한다. 이렇게 최적의 Zone과 lifetime 차이를 반환받은 두 함수 AllocatIOZone(), TakeMigrateZone()은 서로 다른 과정을 거친다.

Append 발생 시 수행되는 AllocateIOZone()은 반환받은 최적의 Zone과 lifetime 차이가 50이나 100이면 비어있는 Zone들 중 한 개를 새로 할당 후 write를 수행한다. 이때 새로 할당된 Zone의 lifetime이 처음으로 write되는 SST File의 lifetime으로 설정된다. 그렇지 않고 lifetime의 차(1~3)가 반환됐다면 해당 Zone에 write를 수행한다.

반면 Migration 발생 시 수행되는 TakeMigrateZone()에서는 비어있는 새로운 Zone을 할당하는 과정 없이 반환받은 최적의 Zone에 write를 수행한다.