creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

LB2-Tree: 키-밸류 SSD에 특화된 인덱스 구조

2021년 2월

서울대학교 대학원

컴퓨터공학부

심 재 훈

LB2-Tree: 키-밸류 SSD에 특화된 인덱스 구조

LB2-Tree: A Index Structure Specialized for Key-Value SSDs

지도교수 김 진 수

이 논문을 공학석사 학위논문으로 제출함
2020년 11월

서울대학교 대학원
컴퓨터공학부
심 재 훈

심재훈의 석사 학위논문을 인준함

2021년 1월

위 원 장	이 재 진	(인)
부위원장	김 진 수	(인)
위 원	Bernhard Egger	(인)

초 록

키-밸류 SSD는 키-밸류 스토어 I/O 스토리지 스택을 구성함에 있어 새로운 패러다임을 제시한다. 키-밸류 SSD는 사용자에게 키-밸류 인터페이스를 제공하여 호스트 시스템 자원의 사용량을 크게 줄이지만 여러 한계점으로 인해 키-밸류 스토어를 대체할 수는 없다. 이 논문에서는 키-밸류 SSD의 특징들을 분석하고 이를 키-밸류 스토어의 대체제로 사용하기 어려운 이유를 설명한다. 또한, 이러한 분석들을 바탕으로 LB2-Tree라는 새로운 인덱스 구조를 제안한다. LB2-Tree는 기존 Copy-On-Write B+Tree에서 나타나는 문제점인 compaction과 cascading update를 키-밸류 SSD의 특성들을 활용해 해결한다. LB2-Tree는 특히 업데이트가 잦은 환경에서 CoW B+Tree에 비해 매우 높은 성능과 매우 낮은 쓰기량을 보인다.

주요어 : 키-밸류 스토어, 키-밸류 SSD, 블록 SSD
학번 : 2019-29590

목 차

제 1 장 서론 1

제2장 키-밸류 SSD 3

2.1 삼성 키-밸류 SSD..... 4

2.2 키-밸류 SSD 성능 분석..... 4

2.2.1 실험 환경..... 5

2.2.2 쓰기 및 읽기 성능..... 5

2.3 키-밸류 SSD 특성 분석..... 7

2.3.1 문자열 키를 성능 예측을 어렵게 한다..... 8

2.3.2 저장된 데이터가 많을수록 읽기 성능은 저하된다..... 9

2.3.3 길이가 긴 키는 성능을 저하한다..... 10

2.4 키-밸류 SSD의 한계..... 10

제3장 트리 데이터 구조 12

3.1 Log-Structured Merge-Tree..... 12

3.2 Log-Structured Merge-Tree의 문제점..... 13

3.3 Copy-On-Write B+Tree..... 15

3.4 Copy-On-Write B+Tree의 문제점..... 15

제4장 Log-structured Blind write B+Tree 17
4.1 개요 17
4.2 쓰기 동작 18
4.3 읽기 동작 20
4.4 로그 병합 작업 21
4.5 가변 길이 노드 쓰기 기법 21
4.6 경량 공간 재확보 기법 22
4.7 멀티 스레드 복구 기법 23

제5장 실험 결과 및 분석 25
5.1 실험 환경 25
5.2 처리량 비교 26
5.3 쓰기량 및 읽기량 비교 27
5.4 복구 속도 비교 29

제6장 결론 31

참고문헌 32

Abstract 34

표 목 차

[표 1] 키-밸류 SSD 실험 환경 ····· 5

[표 2] 실험 환경 ····· 25

[표 3] 워크로드 구성 ····· 26

그 림 목 차

[그림 1] 블록 SSD 기반 키-밸류 스토어와 키-밸류 SSD 간의 I/O 스토리지 스택 비교 ····· 3

[그림 2] 키-밸류 SSD와 블록 SSD의 평균 쓰기 처리량 ····· 6

[그림 3] 키-밸류 SSD와 블록 SSD의 평균 읽기 처리량 ····· 6

[그림 4] 키-밸류 SSD에서 나타나는 read latency spike ·· 8

[그림 5] 키-밸류 SSD 데이터 양에 따른 랜덤 읽기 처리량 9

[그림 6] LSM-Tree의 구조 ····· 13

[그림 7] Write-intensive workload에서의 LSM-Tree의 write latency ····· 14

[그림 8] CoW B+Tree에서의 업데이트 과정 ····· 15

[그림 9] CoW B+Tree에서의 compaction 과정 ····· 16

[그림 10] 노드 B에 대한 재구축 작업 ····· 20

[그림 11] CoW B+Tree에서의 compaction (상), LB2-Tree에서의 compaction (하) ····· 23

[그림 12] CoW B+Tree에서의 복구 진행 방식 ····· 24

[그림 13] LB2-Tree에서의 복구 진행 방식 ····· 24

[그림 14] 각 워크로드별 처리량 비교 ····· 26

[그림 15] 인덱스별 노드 쓰기량 비교 ····· 28

[그림 16] 인덱스 구조별 복구 속도 비교 ····· 29

제 1 장 서론

최근 키-밸류 스토어는 분산 시스템 환경 등에서 많은 사용이 이뤄지고 있다. 키-밸류 스토어들은 기본적인 쓰기, 읽기뿐만 아니라 트랜잭션, 복구, 스냅샷 등의 복잡한 기능들을 사용자에게 제공하여 사용자의 안전한 사용을 보장해준다.

이러한 키-밸류 스토어들은 주로 블록 디바이스 기반으로 이뤄지는데, 블록 디바이스 기반 키-밸류 스토어들은 오래 전부터 여러 문제점들을 보여왔다. 전통적으로 유명한 문제점인 읽기 및 쓰기 amplification에서부터 최근 빠른 차세대 스토리지 디바이스가 출시되면서 호스트 시스템 자원에 대한 오버헤드까지 다양한 연구들이 이러한 문제점들을 해결하는데 초점을 맞추고 있다.

그 중 한가지 해결 방향인 하드웨어로의 키-밸류 기능성 오프로딩은 삼성의 키-밸류 SSD[1], KAML[11], PinK[12] 등 여러 논문들과 제품들이 나오면서 관심을 받고 있다. 하지만 실제 사용자가 이러한 디바이스들을 사용하여 키-밸류 스토어를 대체하기에는 여러 문제점들이 따르는데, 바로 트랜잭션, 복구, 스냅샷 등의 기능이 디바이스에서는 제공되지 않는다는 점이다.

이 논문에서는 키-밸류 SSD에 특화된 LB2-Tree(Log-structured Blind write B+Tree)를 제안하고 이를 키-밸류 SSD 위의 소프트웨어 계층에서의 인덱스로 적용하여 사용자에게 트랜잭션, 복구, 레인지 쿼리, 스냅샷 등의 기능을 제공한다.

이어지는 2장에서는 기존 키-밸류 스토어의 한계점과 키-밸류 스토

리지 디바이스가 등장하게 된 배경에 관해 설명한다. 3장은 삼성 키-밸류 SSD의 특성을 분석하고, 4장에서는 새로운 인덱스 구조인 LB2-Tree의 설계와 구현에 관해 설명한다. 마지막으로 5장에서는 LB2-Tree의 성능을 측정하고 6장에서 논문에 대해 결론짓는다.

제 2 장 삼성 키-밸류 SSD

블록 인터페이스 위에 구현되어 복잡한 스토리지 소프트웨어 스택을 거치고 높은 호스트 시스템 자원을 요구하는 기존 키-밸류 스토어들과는 달리, 키-밸류 SSD는 사용자에게 키-밸류 인터페이스를 제공하여 키를 사용해 디바이스에 데이터를 직접 저장할 수 있게 된다. 간단한 키-밸류 SSD API[그림 1]를 통해 CPU와 메모리와 같은 호스트 시스템의 자원을 적게 쓴다.

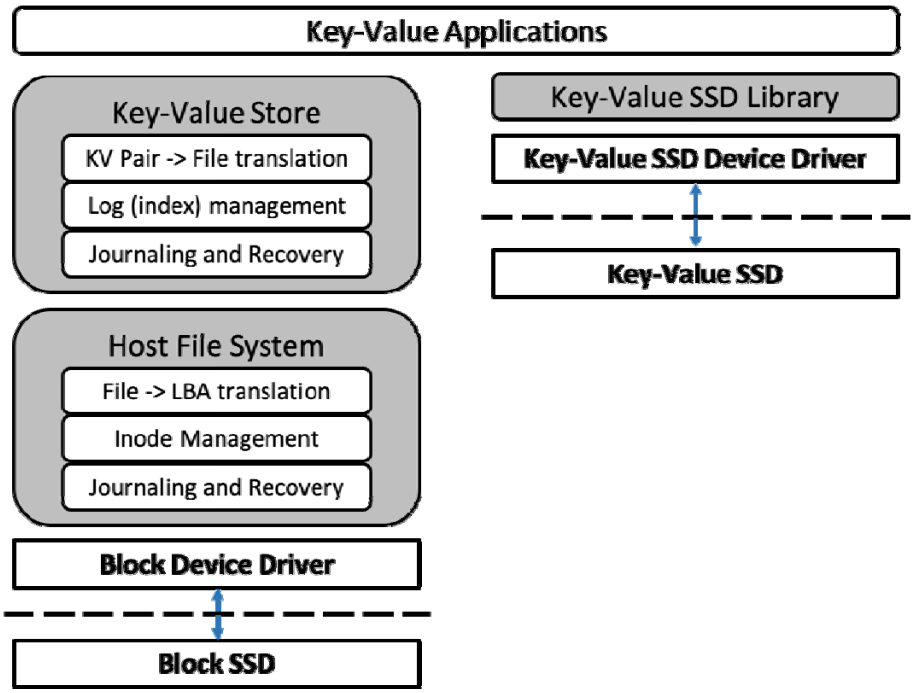

그림 1. 블록 SSD 기반 키-밸류 스토어와 키-밸류 SSD 간의 I/O 스토리지 스택 비교

2.1 삼성 키-밸류 SSD

삼성의 키-밸류 SSD는 제품화된 첫번째 Key-Value SSD로서, 해쉬 테이블을 기반으로 키-밸류 쌍을 관리한다. 디바이스에서 사용되는 키의 크기는 최소 4B에서 255B이고, 밸류의 크기는 최소 0B에서 2MB이다.

키-밸류 SSD 커맨드 셋은 STORE, RETRIEVE, DELETE, EXIST, ITERATE 총 5가지로 표준 NVMe 커맨드 셋을 확장해 만들어졌다. STORE는 호스트가 별도의 매핑을 관리하지 않아도 되게끔 키를 통해 값을 저장하고, 저장된 키-밸류 쌍은 RETRIEVE를 통해 읽어들이고 DELETE를 통해 삭제된다. 이때 STORE는 최대 8개의 write 연산을 하나의 STORE로 합쳐서 적는 batch 커맨드가 지원된다. EXIST 커맨드는 데이터가 키-밸류 SSD에 존재하는지 키를 통해 확인할 수 있게 하고, ITERATE 커맨드는 특정 prefix에 맞는 모든 키들을 탐색한다. 이때 사용자는 키만 가져올 것인지 키-밸류 쌍 모두를 가져올 것인지 정할 수 있으며, ITERATE를 통해 해당 키들을 모두 삭제할 수도 있다.

2.2 키-밸류 SSD 성능 분석

이 절에서는 삼성 키-밸류 SSD의 기본적인 성능을 블록 SSD와 비교해본다. 데이터가 청크로 쓰이는 블록 SSD와 달리, 키-밸류 SSD는 키와 밸류로 나뉘고 키를 통해 키와 밸류를 함께 저장하기 때문에, 다양한 조건 아래에서 실험을 진행하여 블록 SSD와 비교하였다.

2.2.1 실험 환경

[표 1]은 실험 환경 구성에 대해 나타낸 것으로, 현재 키-밸류 SSD는 PM983 NVMe SSD이라는 블록 SSD 위에 KV 펌웨어을 적용시켜 만들어진 것으로, 블록 펌웨어를 적용시킨다면 키-밸류 SSD와 동일한 하드웨어 사양을 지닌 블록 SSD로 사용할 수 있다. 이를 통해 키-밸류 SSD와 블록 SSD 사양 간의 공정한 비교를 진행할 수 있다.

<table>
<tr><td>Device</td><td colspan="3">Samsung PM983 NVMe SSD</td></tr>
<tr><td>SSD Capacity</td><td colspan="3">3.84TB</td></tr>
<tr><td>Kernel</td><td colspan="3">Linux 4.9.5</td></tr>
<tr><td>Device Driver</td><td colspan="3">Kernel Device Driver</td></tr>
<tr><td>KV Firmware</td><td>ETA51KCA</td><td>Block Firmware</td><td>EDA5200Q</td></tr>
<tr><td>키-밸류 SSD Benchmark</td><td>KV_Bench</td><td>Block SSD Benchmark</td><td>fio</td></tr>
</table>

표 1. 키-밸류 SSD 실험 환경

2.2.2 쓰기 및 읽기 성능

키-밸류 SSD의 쓰기 성능을 알아보기 위해 키의 크기를 16B와 128B로 고정시키고 밸류의 크기를 4KB에서 2MB까지 증가시켜가며 블록 SSD와 비교해보았다. 이때 키-밸류 SSD는 삼성 KV API에 함께

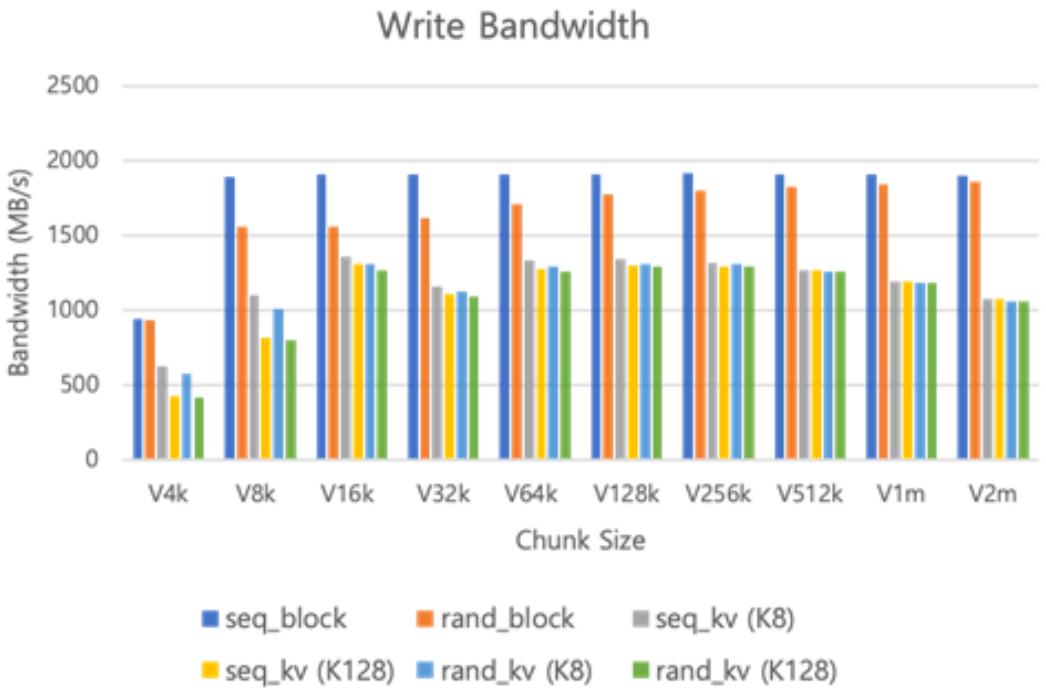

그림 2. 키-밸류 SSD와 블록 SSD의 평균 쓰기 처리량

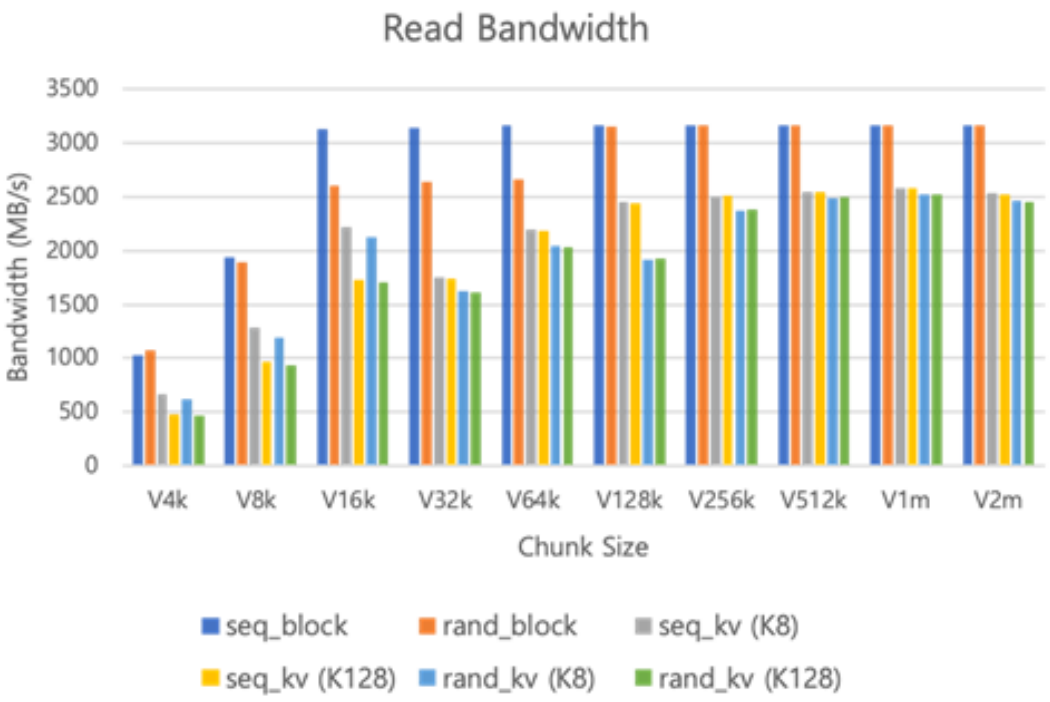

그림 3. 키-밸류 SSD와 블록 SSD의 평균 읽기 처리량

제공되는 KV_Bench를 사용하였고, 블록 SSD의 경우 fio를 사용하였다.

키-밸류 SSD의 평균적인 처리량은 블록 SSD 대비 60% 정도 보이고, 이는 키의 크기가 16B일 때에 비해 128B일 때 더 안좋은 성능을 보였다. 이는 키-밸류 SSD 커맨드에 바로 들어가는 키의 최대 크기가 16B이고, 그 이상일 때는 DMA되기 때문이다. 전체적으로 키-밸류 SSD의 sequential access와 random access간의 성능 격차는 크게 나타나지 않았다.

같은 하드웨어를 사용함에도 키-밸류 SSD가 블록 SSD에 비해 낮은 성능을 보이는 이유는 현재 키-밸류 SSD가 키-밸류 펌웨어를 위해 설계된 하드웨어가 아닌 블록 펌웨어 기반 하드웨어 위에서 동작하는 프로토타입이라는 이유가 크게 작용한다.

2.3 키-밸류 SSD 특성 분석

키-밸류 SSD 성능 분석을 진행하며 사용자가 실제 디바이스를 받기 전까지는 알 수 없는 숨겨진 키-밸류 SSD의 특징들을 분석하였다. 총 3가지로 1) 문자열 키는 성능 예측을 어렵게 한다 2) 저장된 데이터가 많을수록 읽기 성능은 저하된다 3) 길이가 긴 키는 성능을 저하한다.

2.3.1 문자열 키는 성능 예측을 어렵게 한다

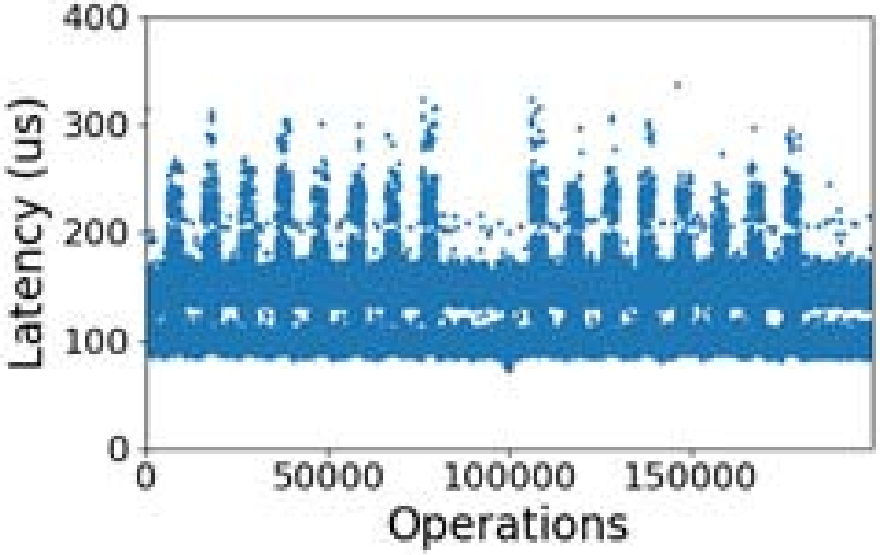

그림 4. 키-밸류 SSD에서 나타나는 read latency spike

키-밸류 SSD에 문자열을 키로 저장하고 그것을 순차적으로 읽을 때 [그림 4]과 같은 원인 불명의 latency spike가 일어났다. 여러 분석을 통해 이러한 latency spike가 일어나는 원인은 키의 마지막 문자가 고정되어 있을 때 나타난다는 것을 알 수 있었다. 문자열에는 언제나 마지막 문자가 널 문자로 고정되어 있어 문자열을 키로 사용할 때 다음과 같은 latency spike가 나타나는 것이었다. 따라서 키-밸류 SSD에 키-밸류 쌍을 저장할 때는 문자열을 키로 사용하는 것은 피하자.

2.3.2 저장된 데이터가 많을수록 읽기 성능은 저하된다

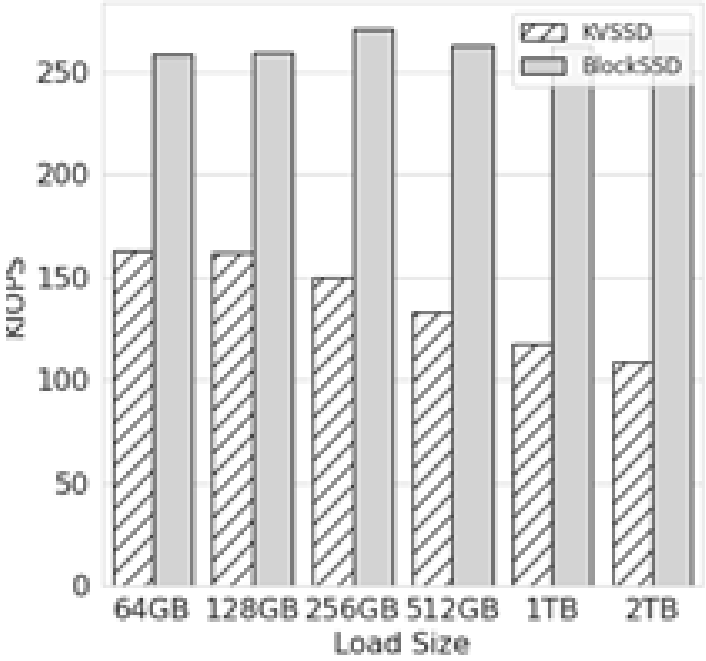

그림 5. 키-밸류 SSD 데이터 양에 따른 랜덤 읽기 처리량

[그림 5]는 64GB에서 2TB까지 키-밸류 SSD에 쓰인 데이터의 양을 변화시켜가며 10분동안 random read를 수행했을 때의 throughput이다. 키-밸류 SSD에 load된 데이터가 많으면 많을수록 성능이 점점 떨어지는 이유는 키-밸류 SSD가 hash table을 기반으로 KV쌍을 관리하기 때문인데, hash table은 저장된 엔트리가 많을수록 collision이 더 많이 일어나 원하는 KV쌍을 찾는데 더 오랜 시간이 걸리게 된다.

2.3.2 길이가 긴 키는 성능을 저하한다

키-밸류 SSD에서 키는 밸류와 함께 NAND 플래시에 저장되기 때문에 작은 키를 사용할수록 키-밸류 SSD에 실제로 적히는 데이터가 적어져 SSD의 수명을 증가시킬 수 있다. 또한 ITERATE 커맨드를 수행할 때, 키-밸류 SSD는 호스트가 할당한 32KB 크기의 버퍼에 키를 적기 때문에 작은 키를 사용할수록 한 번에 가져올 수 있는 키의 수가 많아져 iteration 횟수를 줄일 수 있다.

2.4 키-밸류 SSD의 한계

키-밸류 SSD가 사용자에게 줄 수 있는 장점들은 분명히 있지만, 산업 현장에서의 키-밸류 SSD를 향한 관심도가 낮은 이유는 키-밸류 SSD가 키-밸류 스토어를 온전히 대체하지 못한다는 점이다. 키-밸류 SSD는 현재 단순한 커맨드들만 지원되고 실제 분산 키-밸류 스토어 환경에서 중요한 레인지 쿼리, 트랜잭션, 스냅샷, 복구 등의 기능은 지원되지 않거나 부족하다.

키-밸류 SSD의 ITERATE 커맨드를 사용하여 레인지 쿼리 기능을 비슷하게 제공할 수 있지만, 키의 앞 네 바이트에 대해서만 공통된 키를 가져올 수 있다는 점과 정렬되지 않은 상태로 사용자에게 반환되기 때문에 분명한 한계가 존재한다. 트랜잭션의 경우 키-밸류 SSD는 한 개의 오퍼레이션에 대해서만 원자성을 보장하기 때문에 여러 오퍼레이션에 대해 원자성을 보장해줄 수 없다. 스냅샷의 경우 이전 버전의 데이터를 유지해야 하지만 키-밸류 SSD에 같은 데이터를 쓰면 덮어씌워 져버리기 때문에 스냅샷의 기능을 제공하는 것은 불가능하다. 마지막으로 복구의

경우 로깅 또는 데이터 쓰기 순서 보장을 통해 진행되어야 하지만 키-밸류 SSD에는 블록 SSD처럼 오프셋을 통한 순서 보장이 불가능하기 때문에 복구를 진행할 수 없다.

따라서 키-밸류 SSD가 키-밸류 스토어의 대체제가 되기 위해선 별도의 인덱스를 추가하여 부족한 기능성을 채워줘야 한다.

제 3 장 트리 데이터 구조

이번 장에서는 대표적인 트리 데이터 구조인 LSM-Tree[2]와 Copy-on-Write B+Tree[8]의 특징과 문제점들에 대해 알아본다.

3.1 Log-Structured Merge-Tree

LSM-Tree[2]는 전체적인 구조는 [그림 6]와 같다. LSM-Tree에는 크게 2가지 컴포넌트들이 존재하는데, 메모리에 존재하는 정렬된 데이터 구조인 메모리 컴포넌트와 여러 레벨로 구성되어 있는 디스크 컴포넌트이다. 각 레벨에는 다수의 Sorted String Table(SSTable)라는 정렬된 파일이 존재하게 된다.

메모리 컴포넌트는 사용자가 새로운 키-밸류 쌍을 삽입하였을 때 그것을 우선적으로 받는 쓰기 버퍼로서 동작한다. 이때 지속성을 유지하기 위해 메모리 컴포넌트에 씀과 동시에 disk에 존재하는 Write-ahead Log에도 데이터를 쓴다. 메모리 컴포넌트가 사용자의 삽입을 계속 저장해나가다가 그 크기가 특정 임계값을 넘어서면 상태를 immutable로 변경하고 새롭게 만들어 사용자의 삽입을 수행할 수 있게 한다. Immutable 상태인 메모리 컴포넌트는 SSTable로서 disk에 적히게 되고 level 0에 위치하게 된다. 해당 과정을 flush라고 한다.

Disk에 존재하는 모든 level에는 갖고 있을 수 있는 SSTable 수의 한계가 존재한다. 통상적으로 level i+1은 level i에 비해 2배 더 많은 SSTable들을 보유하게 된다. Flushing을 통해 level 0이 갖고 있을 수

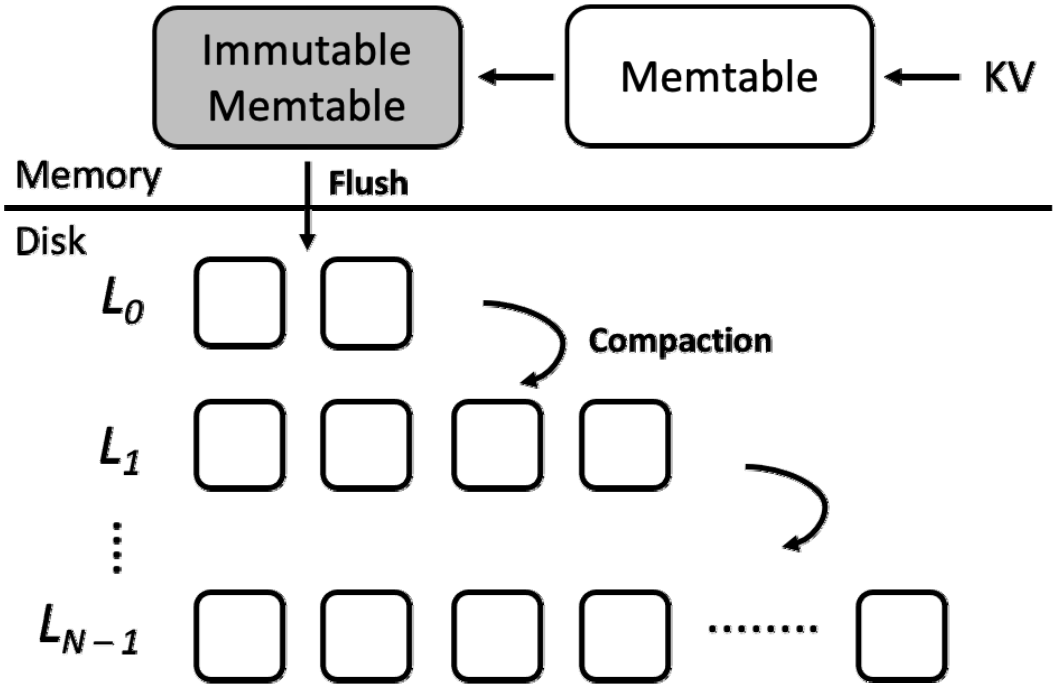

그림 6. LSM-Tree의 구조

있는 SSTable 수를 넘기면 하위 레벨로 SSTable을 merge 시키는데, 이 과정을 compaction이라고 한다. Compaction을 통해 LSM-Tree는 각 레벨의 크기를 유지한다.

3.2 Log-Structured Merge-Tree의 문제점

Compaction을 통해 LSM-Tree는 level을 유지하게 되지만, merge 하는 과정에서 read/write amplification을 발생시킨다. Read/write amplification은 사용자의 요청량 대비 스토리지에서 발생한 요청량으로서, 높을수록 사용자 요청과 무관한 데이터 접근이 많은 것을 의미한다. Read/write amplification이 키-밸류 스토어에 좋지 않은은 영향을 끼치는 이유는 i) 정해진 수명이 있는 SSD의 특징 상, 높은 write

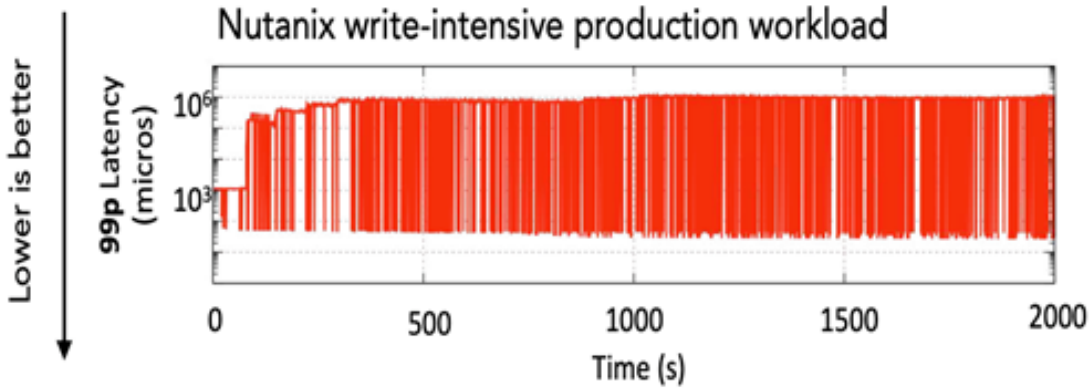

그림 7. Write-intensive workload에서의 LSM-Tree의 write latency. Write latency가 크게 1초까지 걸리는 것을 알 수 있다.[3]

amplification은 SSD의 수명을 깎는다.[10] ii) 사용자의 요청을 위해 사용해야 할 디바이스의 bandwidth를 잡아먹게 되어, 결과적으로 [그림 7]처럼 사용자의 요청을 지연시킨다.[3].

최근에는 OptaneSSD[4], Non-Volatile Memory[5, 6] 등의 빠른 스토리지 디바이스들이 등장하면서 LSM-Tree의 문제점으로 CPU overhead가 부상하고 있다.[7] 새로운 스토리지 디바이스들이 높은 throughput과 낮은 latency를 보이면서 LSM-Tree로는 디바이스 bandwidth를 모두 활용하지 못하고 있고 있는데, 그 이유로는 compaction와 같은 작업들이 CPU와 I/O를 많이 잡아먹기 때문이다. 과거 느린 디바이스를 사용할 때는 sequential access와 random access의 성능 차이가 컸기 때문에 모든 데이터를 정렬시키는 것이 중요했지만 최근 디바이스는 sequential access와 큰 차이를 보이지 않는 random access와 매우 낮은 latency를 보이기 때문에 디바이스 bandwidth를 더 활용하기 위해 CPU를 희생하여 데이터를 정렬하고 compaction 작업을 진행한 것이 이제는 오버헤드로 바뀌었다.

3.3 Copy-On-Write B+Tree

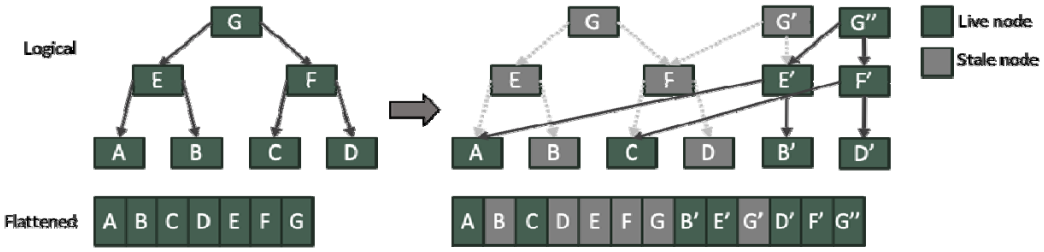

그림 8. CoW B+Tree에서의 업데이트 과정

Copy-On-Write B+Tree[8]는 B+Tree의 일종으로서, 새로운 쓰기 요청이 들어오면 그것을 파일의 끝에 추가하는 append-only 방식으로 수행한다. [그림 8]에서 노드 B가 수정되면 수정된 노드 B' 을 기존 위치에 덮어쓰지 않고 파일의 끝에 추가하게 된다. 또한, 노드의 위치가 변경되었기 때문에 노드 B에서부터 루트 노드까지 모두 데이터가 수정되어야 한다. 결과적으로 부모 노드인 E와 G 또한 E' 와 G' 으로 수정되어 파일의 끝에 쓰이게 된다. 이러한 append-only 방식으로 인해 CoW B+Tree는 다중 버전 동시성 제어와 경량 스냅샷을 제공하는데 특화되어 있다.

3.4 Copy-On-Write B+Tree의 문제점

leaf 노드에 데이터를 추가할 때 해당 노드를 블록 단위만큼 읽고 추가되는 데이터의 양과 무관하게 다시 블록 단위만큼 써야 하는 점은 모든 B+Tree들이 가진 문제점이다. 이는 B+Tree들이 블록 기반 디바이스에 노드들을 저장하기 때문에 발생하고, leaf node에 1개의 엔트리만 변경되어도 leaf node가 블록 단위만큼 새로 쓰여야 하기 때문에 균등

분포 워크로드에서 더 높은 write amplification을 보인다.

CoW B+Tree에서의 쓰기는 모두 append-only 방식으로 이뤄지기 때문에 한 개의 노드를 수정하는 것이 루트 노드까지의 수정을 초래하는 “wandering tree” 문제[13]를 발생시킨다. 이는 높은 쓰기 amplification을 발생시킨다. 이를 완화하기 위해 CoW B+Tree들은 주로 쓰기 요청을 모아놨다가 batch로 처리를 하지만 문제를 완전히 해결해주지는 못한다.

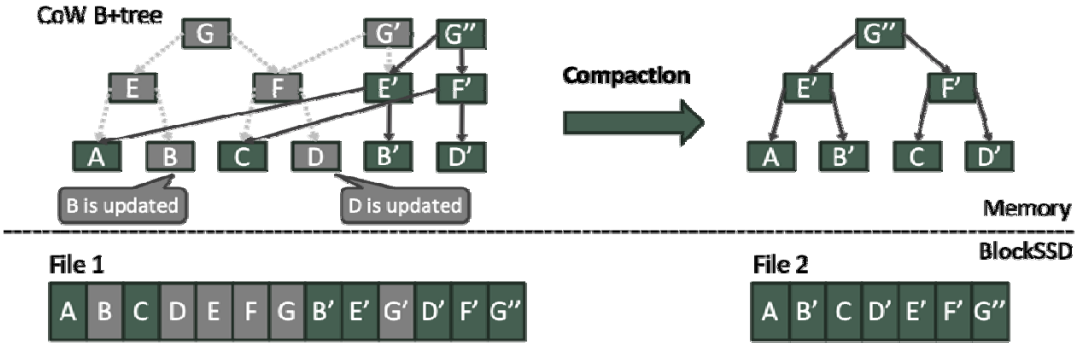

그림 9. CoW B+Tree에서의 compaction 과정

또한, Append-only 방식은 유효하지 않은 오래된 데이터를 생성하고 시간이 지남에 따라 파일 크기를 증가시키기 때문에 주기적인 데이터 재확보 작업이 필요하다. 따라서 CoW B+Tree는 compaction이라는 작업을 수행한다. Compaction 작업은 키 순서로 tree를 탐색하여 유효한 키-밸류 쌍을 새로운 파일로 옮기고, 새롭게 B+Tree를 구축한 뒤 이전 파일을 제거하여 사용 가능한 공간을 확보한다. [그림 9]처럼 유효한 데이터를 모두 새 파일에 다시 쓰기 때문에 대용량 디스크 I/O가 발생하며 전체 성능에 큰 영향을 미친다.

제 4 장 Log-structured Blind write B+Tree

이번 장에서는 키-밸류 SSD에 부족한 기능들을 지원해주기 위해 소프트웨어 계층에 적용하는 인덱스 구조인 Log-structured Blind write B+-Tree (LB2-Tree)를 소개한다.

4.1 개요

키-밸류 SSD에는 이미 키-밸류 쌍이 저장되기 때문에 인덱싱을 위해서는 LSM-Tree보다는 CoW B+Tree가 더 적합하다. 따라서 이 논문에서는 CoW B+Tree를 키-밸류 SSD에 맞게 특화시킨 LB2-Tree를 소개한다. LB2-Tree에서의 노드들은 키-밸류 SSD에 각각 키-밸류 쌍으로 저장되고 각 노드 안의 엔트리는 (키, 키-밸류 SSD 키)로 이뤄져있고 이때 키-밸류 SSD 키는 키-밸류 SSD에 저장될 때 사용된 키이다.

노드를 수정할 때 발생하는 read-modify-write 문제와 wandering tree problem을 해결하기 위해 LB2-Tree는 수정된 데이터를 해당 leaf node에 대응하는 log에 적음으로서 쓰기 동작 중에 발생하는 추가적인 디스크 읽기와 쓰기를 효과적으로 줄인다.

4.2 쓰기 동작

[알고리즘 1]에서처럼 쓰기는 타겟 리프 노드를 찾은 뒤 시작된다. 이때 기존의 CoW B+Tree의 경우 타겟 리프 노드가 메모리에 없다면 디스크로부터 읽어 들인 뒤 쓰기를 수행하지만, LB2-Tree는 리프 노드가 메모리에 없을 때 키-밸류 SSD로부터 노드를 읽지 않고 삽입되는 엔트리를 하나의 키-밸류 쌍으로 저장을 한다. 이를 통해 노드 한개 크기만큼의 읽기와 쓰기를 엔트리 하나만큼의 쓰기로 줄일 수 있고 노드가 다시 쓰이는 것이 아닌 엔트리가 로그로 저장되기 때문에 wandering tree problem 또한 피할 수 있다.

리프 노드가 메모리에 있다면 수행하는 쓰기 동작이 새로운 엔트리의 삽입인지 기존 엔트리의 수정인지 확인한 후 수정이라면 수정되는 데이터만을 하나의 키-밸류 쌍으로만 저장한다. 새로운 엔트리의 삽입이라면 부모 노드의 엔트리도 바뀔 가능성이 존재하지만, 기존 엔트리의 수정이라면 리프 노드만 수정됐는데도 루트 노드까지 수정이 강요되기 때문에 이를 피하고자 기존 엔트리의 수정 또한 로그로 저장한다. 루트 노드까지의 수정을 피함으로써 불필요한 인덱스 쓰기를 줄일 뿐만 아니라, 새로 쓰이는 노드들이 모두 새로운 위치로 바뀌기 때문에 캐시를 필요 이상으로 비대하게 채우는 일 또한 피할 수 있게 된다.

Algorithm 1: LB2-Tree Insertion

```
Input  : k: key, v: value, r: root
Output : SUCCESS/FAILURE
leaf ← find_leaf (r,k);
if Leaf Node is in memory then
    if k exists in Leaf Node then
        /* This operation is update                    */
        add_log (leaf,k,v) ;
    else
        /* This operation is insert                    */
        add_entry (leaf,k,v) ;
        cascade_update_to_root (leaf) ;
    end
else
    add_log (leaf,k,v) ;
end
return SUCCESS;
```

알고리즘 1. LB2-Tree 쓰기 동작

Algorithm 2: LB2-Tree Leaf Node Read

```
Input  : k: key, r: root
Output : Address of Leaf Node
leaf ← find_leaf (r,k);
if Leaf Node has logs then
    /* Need Leaf Node reconstruction                   */
    log_list ← collect_all_logs (leaf) ;
    merge_all_logs (leaf,log_list) ;
end
return leaf
```

알고리즘 2. LB2-Tree 리프 노드 동작

4.3 읽기 동작

읽기 동작의 경우 모든 것이 CoW B+Tree와 같게 처리된다. 루트 노드에서부터 특정 키에 대해 탐색을 진행하면서 리프 노드에 도달하면 키에 대해서 찾고 올바른 키-밸류 쌍을 리턴한다

하지만 LB2-Tree의 경우 업데이트 또는 노드가 메모리가 없을 때 저장을 로그를 통해 하기 때문에 CoW B+Tree와는 달리 리프 노드 재구축 작업이 필요하다. [알고리즘 2]에서 리프 노드를 찾은 뒤, 해당 리프 노드에 로그가 존재하는지를 확인한다. 해당 노드에 로그가 존재한다면 리프 노드 읽기 과정에서 추가적인 재구축 과정이 포함된다. [그림 10]에서처럼 노드 B에 로그가 존재한다면 키-밸류 SSD의 ITERATE 커맨드를 이용해서 해당 리프 노드에 대한 모든 로그를 수집한 뒤, 유효한 로그들만 남겨둔 채 이를 리프 노드에 차례로 병합해서 유효한 리프 노드를 생성하게 된다.

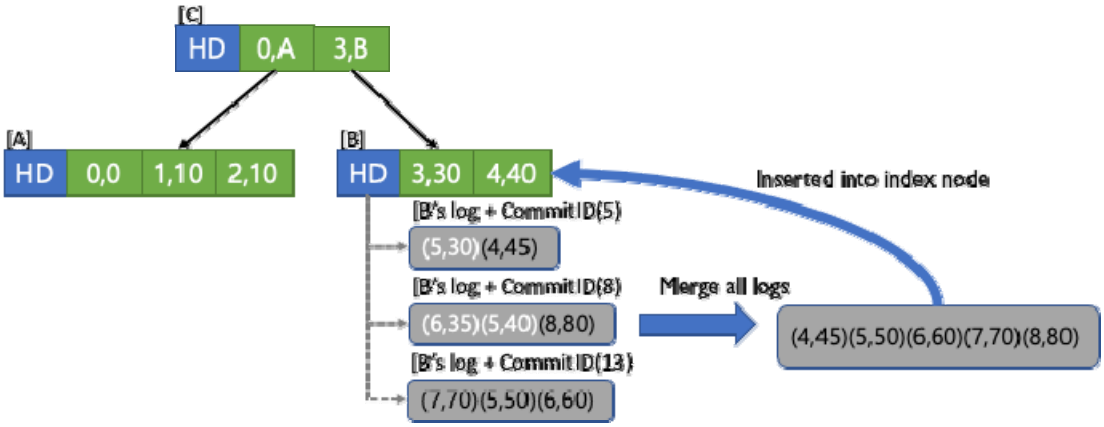

그림 10. 노드 B에 대한 재구축 작업

4.4 로그 병합 작업

로그가 존재하는 리프 노드를 읽어 들이는 과정은 온전한 리프 노드를 읽어 들이는 것보다 시간이 더 소요된다. 또한, 로그가 많이 존재할수록 키-밸류 SSD의 공간을 많이 차지하게 된다. 따라서 LB2-Tree는 리프 노드에 적힌 로그의 수를 관리하다 그 수가 특정 임계값을 넘어갈 때 해당 리프 노드에 대한 로그 병합 작업을 수행한다.

로그 병합 과정은 위에서 설명된 과정을 통해 해당 리프 노드를 읽어 들인 뒤 재구축을 진행하여 만들어진 온전한 노드를 새롭게 키-밸류 SSD에 쓰고 기존 리프 노드와 해당하는 로그들을 모두 삭제한다. 이를 통해 키-밸류 SSD의 공간을 효율적으로 쓰고 읽기 동작 레이턴시를 줄일 수 있게 된다.

로그의 수에 대한 임계값에 따라 쓰기-읽기 및 공간 효율성 trade-off가 발생하는데, 임계값이 크면 병합 작업 없이 로그를 쓰기만 하기 때문에 쓰기 성능이 높아지는 반면 읽기를 수행할 때 재구축 작업을 자주 해야되기 때문에 읽기 성능이 저하되고, 임계값이 낮으면 자주 로그 병학 작업이 수행되기 때문에 쓰기 성능이 저하되는 반면 온전한 리프 노드들이 많아지므로 읽기 성능과 키-밸류 SSD 공간 효율성이 높아진다. 현재 LB2-Tree는 임계값으로 4K 크기의 노드에 들어갈 수 있는 최대 엔트리 수로 고정해놓았다.

4.5 가별 길이 노드 쓰기 기법

블록SSD기반 CoW B+Tree의 경우 디바이스에 읽고 쓰는 단위가 블록이기 때문에 통상적으로 인덱스 노드의 크기를 4K로 고정하고 4K 단

위로 읽고 쓴다. 이 과정 중에 노드가 꽉 차지 않음에도 불구하고 4K만큼 쓰기를 수행해야하기 때문에 불필요한 쓰기가 발생되지만 이는 블록 인터페이스를 사용하는 이상 어쩔 수 없었다.

하지만 LB2-Tree는 키-밸류 인터페이스를 사용하기 때문에 사용자에게 가변 길이 키-밸류 쌍 쓰기 및 읽기 기능을 제공한다. 따라서 LB2-Tree에서는 노드의 실제 크기만큼만 읽고 쓴다. CoW B+Tree 또한 노드의 크기 단계를 나눠서 노드를 저장할 수 있지만 그에 따라 각 노드별 크기를 별도로 관리해야한다. 하지만 LB2-Tree의 경우 키-밸류 쌍의 크기를 키-밸류 SSD에서 관리해주기 때문에 사용자는 별도의 크기 관리 없이 키에 대해서 읽기 요청만 수행하면 된다.

4.6 경량 공간 재확보 기법

기존 CoW B+tree는 데이터를 파일의 끝에 로깅하기 때문에 사용함에 따라 유효하지 않은 데이터들이 생겨난다. 파일의 크기가 너무 커지면 이를 정리하기 위해 compaction을 진행하는데 [그림 9]과 같이 file1을 스캔한 후 file2를 다시 쓰기 때문에 LSM-Tree의 compaction과 마찬가지로 높은 CPU 사용량과 I/O overhead를 초래한다.

하지만 LB2-Tree는 CoW B+Tree와 상황이 다르다. 연속적인 공간에 블록 단위로 순차적으로 쓰는 append-only 방식과는 달리, 디바이스에 모든 것 키로 저장하기 때문에 LB2-Tree는 space reclaim 동작이 따로 필요하지 않고 단순히 해당 키를 삭제만 해주면 된다. 따라서 LB2-Tree는 [그림 11]에서처럼 stale 처리된 데이터에 대해서 그 키를 갖고 있다가 stale 데이터의 양이 특정 임계값을 넘으면 해당 데이터들에 대해 백그라운드 비동기식 삭제를 통해 stale 데이터들을 모두 삭

제한다. 이 방식은 stale 영역을 추적하기 위해 메모리에 유지하고 지속성을 보장해줘야 하는 데이터의 양이 매우 적어지고 호스트와 디바이스 간의 데이터 이동이 존재하지 않는다.

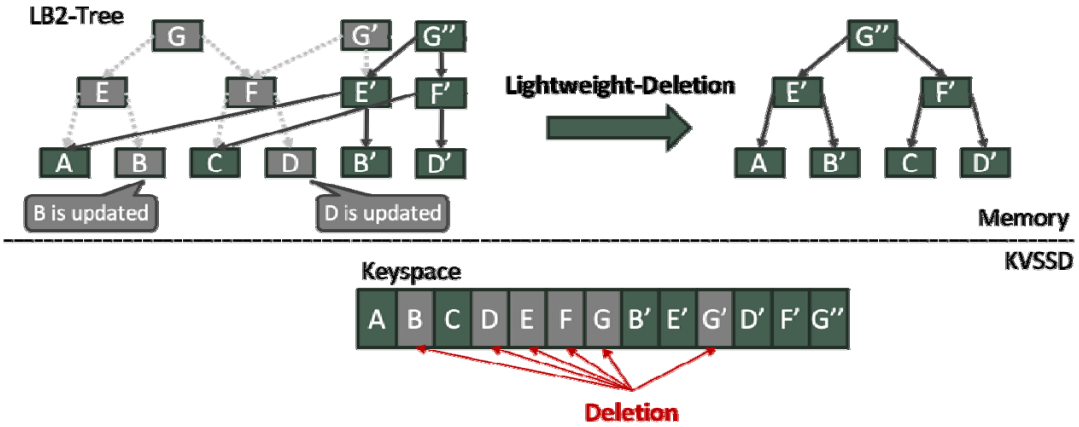

그림 11. LB2-Tree에서의 compaction 과정

4.7 멀티 스레드 복구 기법

실행 중 crash가 발생할 때 CoW B+Tree는 [그림 12]처럼 인덱스들을 가장 마지막으로 저장한 위치에서부터 앞으로 탐색해나가면서 아직 인덱스에 저장되지 않은 데이터들을 복구해나간다. 이 작업은 연속적인 블록 공간에서 특정 위치에서부터 앞으로 스캔하면서 복구를 진행하기 때문에 멀티 스레드로 동작할 수 없다.

하지만 LB2-Tree는 연속적인 블록 공간을 사용하지 않고, 커밋 ID와 현재 커밋 안에서의 번호를 합친 것을 디바이스에 적는 키로 사용하여 저장한다. 저장되는 모든 데이터는 독립적이기 때문에 LB2-Tree는 [그림 13]처럼 커밋 ID별로 ITERATE 커맨드를 수행하는 스레드를 생

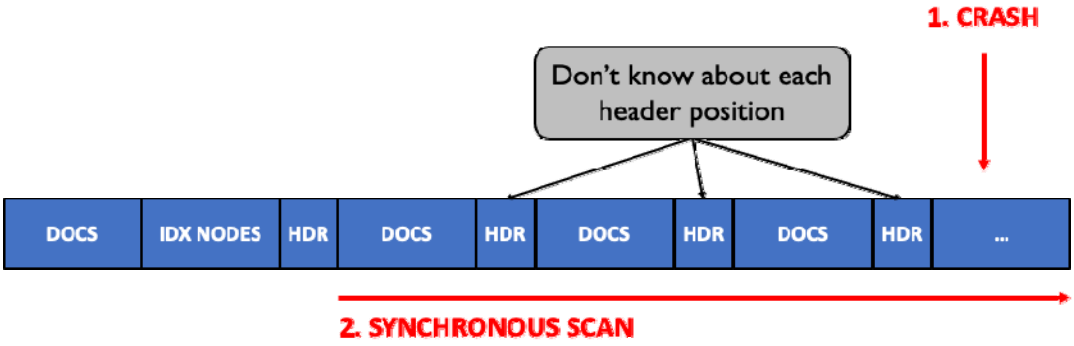

그림 12. CoW B+Tree에서의 복구 진행 방식

성하고 각 스레드는 해당 커밋 ID에 속한 데이터들을 비동기식으로 처리하여 빠르게 복구를 진행할 수 있게 된다.

그림 13. LB2-Tree에서의 복구 진행 방식

제 5 장 실험 결과 및 분석

이번 장에서는 4장에서 설계한 LB2-Tree의 성능을 블록 SSD 기반 CoW B+Tree와 비교해본다. 실험에서 알고자 하는 것은 다음과 같다. 1) 블록 SSD에 비해 상대적으로 떨어지는 성능의 키-밸류 SSD를 사용함에도 불구하고 LB2-Tree가 CoW B+Tree와 유사한 성능을 보여주는가? 2) 쓰기량과 읽기량은 줄어드는가? 4) 멀티 스레드 복구 기법는 의미있는 성능을 보여주는가.

5.1 실험 환경

Processor	Intel i7-9700K CPU @ 3.60GHz
Memory	64GB
OS	Ubuntu 16.04 LTS Kernel 4.9.5
GCC	5.5.0

표 2. 실험 환경

실험에 사용된 시스템 사양은 위 [표 2]과 같다. CoW B+Tree와 LB2-Tree 모두 같은 머신과 같은 SSD 위에서 동작했으며, 이때 사용된 SSD 구성은 [표 1]와 같다.

LB2-Tree의 성능 평가는 NoSQL 벤치마크인 Yahoo! Cloud Serving Benchmark(YCSB)[9]를 사용하여 진행하였다. 또한 update

를 효과적으로 처리하는지 확인하기 위해 update 100%로 이뤄진 워크로드를 추가하였다.

	YCSB A	YCSB B	YCSB C	Update
R:W ratio	50:50	95:5	100:0	0:100
Query type	Point			
Request distribution	Zipfian			

표 3. 워크로드 구성

5.2 처리량 비교

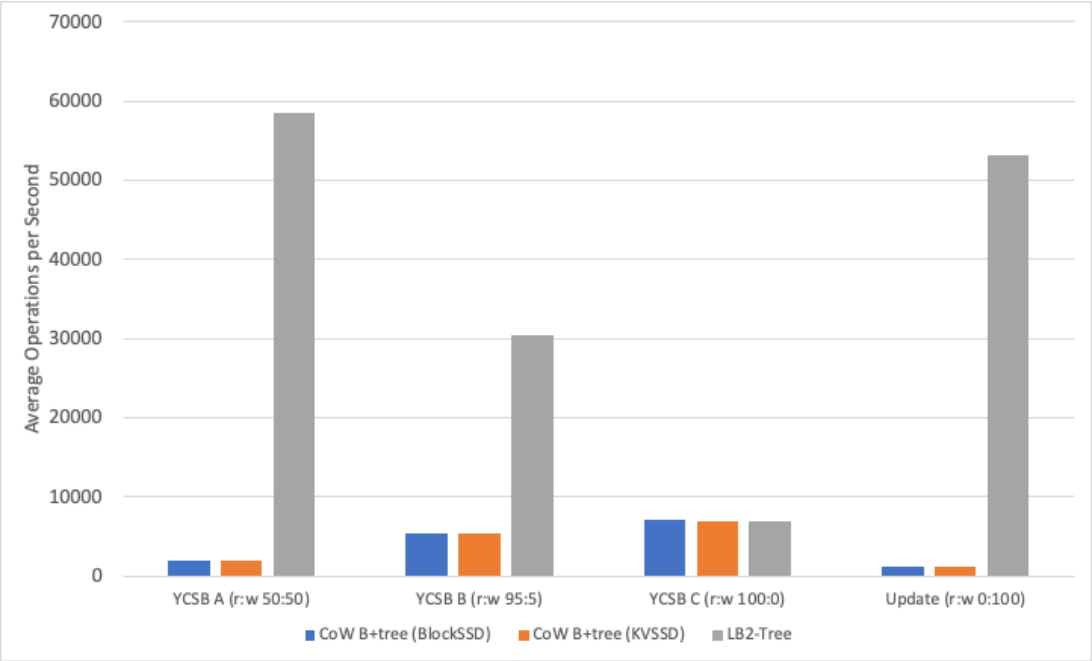

그림 14. 각 워크로드별 처리량 비교

LB2-Tree의 성능 비교는 블록 SSD기반 CoW B+Tree와 그것을 키-밸류 SSD로 포팅한 키-밸류 SSD 기반 CoW B+Tree와 진행된다. 실험 환경을 다음과 같다. 1M개의 키-밸류 쌍을 삽입한 후, 1M개의 키에 대해 zipfian distribution으로 워크로드를 수행하였다. Cache의 크기는 모두 40MB로 고정되어 있고, 4K개의 오퍼레이션마다 노드들을 디스크에 저장하는 커밋이 이뤄졌다.

실험 결과, LB2-Tree는 update가 섞인 모든 워크로드에서 나머지 두 인덱스 구조보다 월등히 좋은 성능을 보였다. 그 이유는 LB2-Tree는 별도의 디스크 읽기를 수행하지 않고 바로 update를 저장하기 때문에 synchronous read를 수행함에 따라 오는 성능 저하를 피하고, 루트 노드까지의 업데이트가 수행되지 않기 때문이다.

읽기 성능 또한 CoW B+Tree와 비슷한 성능을 보이는데 이는 캐시에서 노드를 충분히 보관하고 있어서 재구축 과정에 있어 오는 성능 저하를 최대한 피할 수 있기 때문이다. 현재 실험 구성에서 캐시의 크기는 충분히 현실적인 크기로 구성되어 있지만, 캐시의 크기가 작아진다면 디스크로부터 읽어야 하는 노드의 수가 많아지고, 한 개의 노드만 읽으면 되는 CoW B+Tree와는 달리 LB2-Tree는 노드를 읽고 로그들을 수집하여 병합하는 과정을 거쳐야 해서 성능이 저하되게 된다.

5.3 쓰기량 및 읽기량 비교

[그림 15]는 각 워크로드를 수행하며 발생한 인덱스 노드 쓰기량을 비교한 그래프이다. 세 가지 워크로드 모두에서 LB2-Tree가 나머지 두 인덱스 구조보다 매우 낮은 인덱스 노드 쓰기량을 보이는데, 이는 크게 4가지 이유가 존재한다. i) 로깅을 통해 수정되는 엔트리만 기록하기

때문에 노드 전체를 다시 쓰지 않아도 된다 ii) 캐시에 없는 노드에 대해 쓰기를 수행하기 위해 노드를 읽고 그것이 캐시에 쓰임에 따른 캐시 eviction이 발생하지 않는다 iii) 리프 노드에서 발생한 쓰기에 따른 루트 노드까지의 cascading update가 발생하지 않는다 iiii) 가변 길이 노드 쓰기 기법에 따라 노드를 쓸 때 블록 단위로 쓰지 않고 실제 노드 크기만큼만 저장된다.

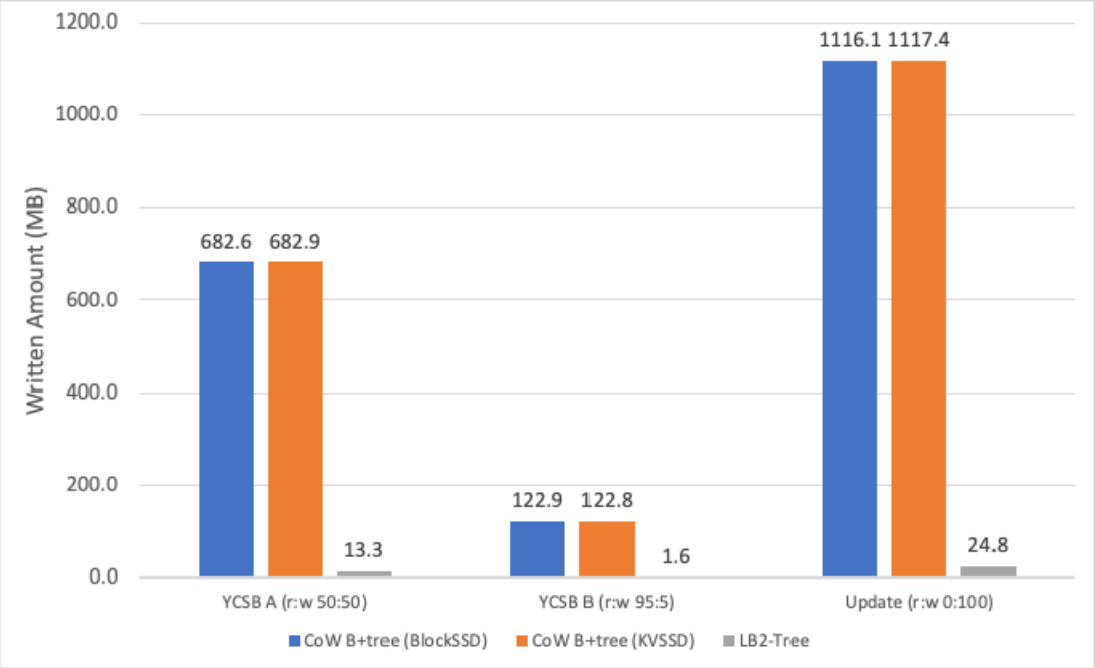

그림 15. 인덱스 구조별 인덱스 노드 쓰기량 비교

읽기량의 경우 CoW B+Tree는 높은 읽기량이 나타났지만, LB2-Tree는 위의 세 가지 워크로드 모두에서 발생하지 않았다. 이는 LB2-Tree가 캐시를 더 효율적으로 쓰기 때문인데, CoW B+Tree의 경우 리프 노드에서 루트 노드까지 모두 다시 파일에 쓰일 때 새로운 위치에 쓰이기 때문에 캐시에도 새로운 블록으로 쓰이게 된다. 따라서 캐시에서 퇴출당하는 유효한 노드들이 많아진다. 반면에 LB2-Tree의 경

우 리프 노드에서 발생한 업데이트가 로그로만 저장되고 새로운 위치에 노드를 다시 쓸 필요가 없다. 따라서 이미 캐싱된 위치에 덮어 쓰이고 불필요한 노드 퇴출이 발생하지 않는다.

5.4 복구 속도 비교

복구 속도 비교는 YCSB A 워크로드를 수행하다가 crash가 발생했을 때의 복구 속도를 비교하였다. [그림 16]에 나타난 각 인덱스 구조별 복구 시간은 최악의 복구 시간을 나타낸다.

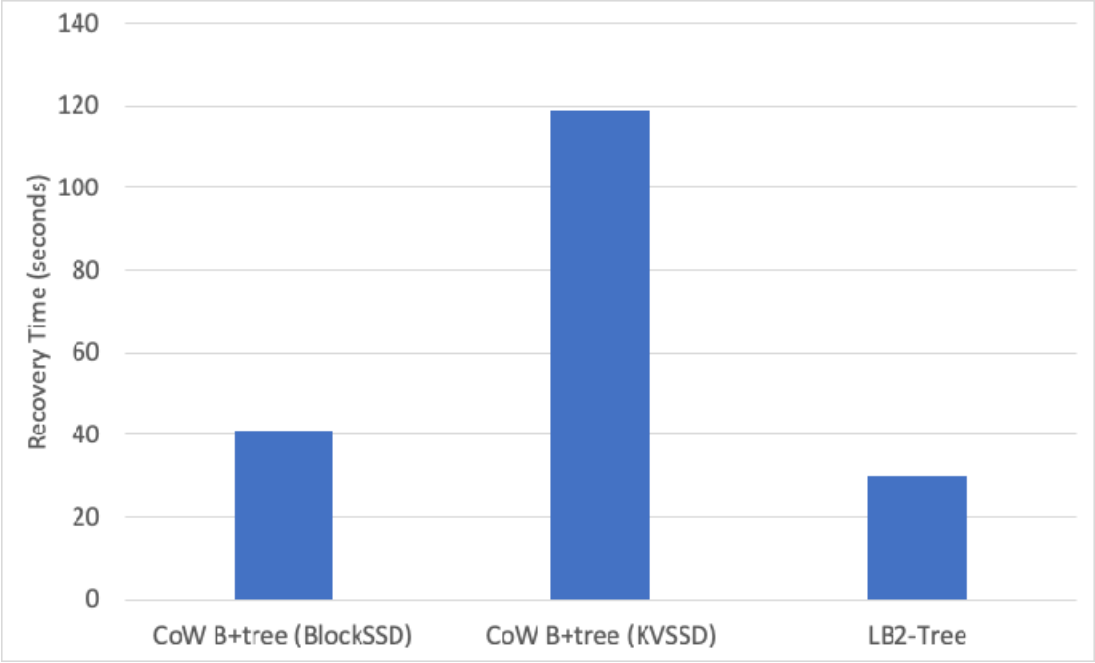

그림 16. 인덱스 구조별 복구 속도 비교

CoW B+Tree는 마지막으로 인덱스 노드를 저장한 위치에서부터 마지막 파일 위치까지 동기적으로 스캔을 수행해가며 복구를 진행한다. 이때 키-밸류 SSD의 기본 성능이 블록 SSD보다 떨어지기 때문에 키-밸

류 SSD 기반 CoW B+Tree는 블록 SSD 기반 CoW B+Tree보다 매우 낮은 복구 속도를 보인다. 하지만 LB2-Tree의 경우 멀티 스레드 복구 기법을 이용하여 여러 스레드가 각각 커밋 위치에 대해 비동기적으로 복구를 수행하기 때문에 키-밸류 SSD의 느린 읽기 성능을 극복하고 키-밸류 SSD 기반 CoW B+Tree에 비해 4배 더 빠른 속도를 보이고 블록 SSD 기반 CoW B+Tree 보다 좋은 성능을 보인다.

제 6 장 결론

본 연구에서는 키-밸류 SSD의 성능과 특성을 분석하고, 키-밸류 SSD가 트랜잭션, 복구, 레인지 쿼리, 스냅샷 등의 기능을 제공하는 데 있어 부족하여 키-밸류 스토어를 완전히 대체하지 못함을 알아냈다. 따라서 키-밸류 SSD에 인덱스 구조를 키-밸류 SSD 위에 올려 사용자에게 키-밸류 스토어로서의 기능들을 제공해주고자 하였다.

LB2-Tree는 블록SSD 기반 CoW B+Tree를 키-밸류 SSD에 특화시킨 것으로, CoW B+Tree의 장점은 취하고 단점들은 키-밸류 SSD의 특성을 사용하여 해결하였다. LB2-Tree는 블록SSD 기반 CoW B+Tree보다 좋거나 비슷한 성능을 보임과 동시에 매우 낮은 읽기량 및 쓰기량과 낮은 호스트 시스템 자원 사용량을 보인다.

참고 문헌

[1] Kang, Yangwook, et al. "Towards building a high-performance, scale-in key-value storage system." Proceedings of the 12th ACM International Conference on Systems and Storage. 2019.

[2] O' Neil, Patrick, et al. "The log-structured merge-tree (LSM-tree)." Acta Informatica 33.4 (1996): 351-385.

[3] Balmau, Oana, et al. "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores." 2019 USENIX Annual Technical Conference (USENIX ATC 19). 2019.

[4] Intel Optane SSD. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html.

[5] Intel optane dc persistent memory architecture overview. https://techfieldday.com/video/intel-optane-dc-persistent-memory-architecture-overview/

[6] Raoux, Simone, et al. "Phase-change random access memory: A scalable technology." IBM Journal of Research and Development 52.4.5 (2008): 465-479.

[7] Lepers, Baptiste, et al. "KVell: the design and implementation of a fast persistent key-value store." Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019.

[8] Becker, Bruno, et al. "An asymptotically optimal multiversion B-tree." The VLDB Journal 5.4 (1996): 264-275.

[9] Cooper, Brian F., et al. "Benchmarking cloud serving systems with YCSB." Proceedings of the 1st ACM symposium on Cloud computing. 2010.

[10] Lu, Lanyue, et al. "WiscKey: Separating Keys from Values in SSD-conscious Storage." 14th USENIX Conference on File and Storage Technologies (FAST 16). 2016.

[11] Jin, Yanqin, et al. "KAML: A flexible, high-performance key-value SSD." 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2017.

[12] Im, Junsu, et al. "PinK: High-speed In-storage Key-value Store with Bounded Tails." 2020 USENIX Annual Technical Conference (USENIX ATC 20). 2020.

[13] A. B. Bityutskiy. JFFS3 design issues. http://www.linux-mtd.infradead.org, 2005.

Abstract

LB2-Tree: A Index Structure Specialized for Key-Value SSDs

Jae-Hoon Shim
Department of Computer Science & Engineering
The Graduate School
Seoul National University

Key-Value SSDs presents a new paradigm in constructing the Key-Value Store I/O storage stack. Key-Value SSDs provide users with a key-value interface, which significantly reduces the use of host system resources, but can not replace Key-Value stores due to multiple limitations. This paper analyzes the characteristics of Key-Value SSDs and explains why they can not be used as an alternative to Key-Value stores. Based on these analyses, this paper propose a new index structure called LB2-Tree. LB2-Tree is based on Copy-on-Write B+Tree and solves problems like compaction, high write amplification, high CPU overhead, and cascading updates by using the characteristics of a Key-Value SSDs. LB2-Tree shows high

throughput and low write and read amplification compared to CoW B+tree, especially in update-intensive workloads.

Keywords : Key-Value Store, Key-Value SSD, Block SSD
Student Number : 2019-29590