보 고 서

컴퓨터공학 종합설계

LSM-TFL 을 이용한 SSD 성능 향상

TEAM LKM

12121550 임 준수 (angkswnstn@gmail.com)

12120315 김 근표 (kgp1202@naver.com)

12101451 문 한별 (dynamichb67@gmail.com)

1. 목차

1. 주제 정의

기존의 SSD device의 S/W 구조보다 읽기, 쓰기, 삭제 연산시의 성능 및 디스크 공간에 대한 효율이 좋은 SSD device S/W 구조를 개발한다.

2. 주제 선정 배경

2-1. 주제 선정 동기

현재 시중에서 사용되는 대부분의 컴퓨터는 file system이 HDD에 적합하도록 설계되어 있다. 따라서 SSD에 맞는 file system을 설계해서 모든 컴퓨터에 이식시키는 것은 실질적으로 비용이 많이 들 수 밖에 없다. 그러나 HDD에 맞는 파일 입출력 인터페이스를 맞추게 되면 SSD를 기존 file system에서 사용 가능하다. SSD의 물리적 특성 상 디스크 공간 내에 overwrite가 되지 않기 때문에 삭제 연산 후 빈 공간들에 대한 garbage collection 등의 기능이 필요하게 된다. 이 기능을 수행하는 것이 SSD 내의 FTL이다.

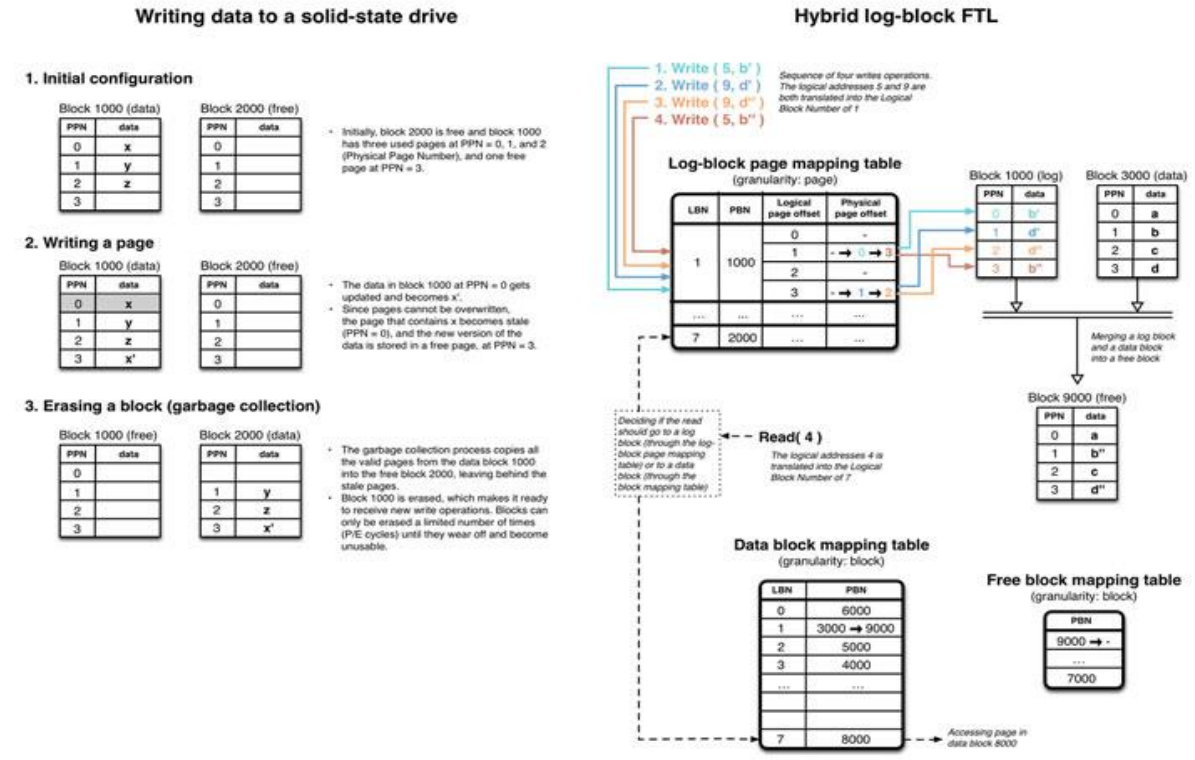
기존의 SSD에서 사용되는 FTL은 page table을 이용한 방식으로, 간단하지만 여러 문제점을 가지고 있다.
첫째로는 주소 변환을 위한 mapping table이 필요하다. 둘째로는 디스크 연산 중 삭제 연산의 경우는 읽기, 쓰기와의 연산단위와 다르기 때문에 그것을 관리하기 위한 garbage collection이 필요하다는 점이 있다. 마지막으로 SSD의 물리적 구조가 flash device를 이용하였으므로 수명이 제한적이며, 따라서 여러 적절한 공간에 데이터를 알맞게 저장하는 wear leveling을 필요로 한다는 점이다. 당연히 필요하다고 생각되지만 메모리 용량 및 디스크 용량에 차이가 있을 수 있고 성능에도 상당한 불리함이 있을 것이라 생각된다. 따라서 SSD의 내부 소프트웨어 구조를 변경하여 위의 기능들을 줄이면서 같은 효과를 낼 수 있는 방안을 생각해 내어, SSD의 성능을 향상할 수 있을 것이라 예상한다.

구체적으로 설명하자면, 기존의 FTL은 각 논리 주소-물리 주소 변환을 위한 mapping table이 필요하다. 이 공간은 meta-data로 이용이 되고, 디스크 영역의 일정 공간을 차지하게 된다. 따라서 실질적인 정보를 저장하는 DISK로 사용되지 못하고 이것은 곧 사용자가 사용할 수 있는 공간이

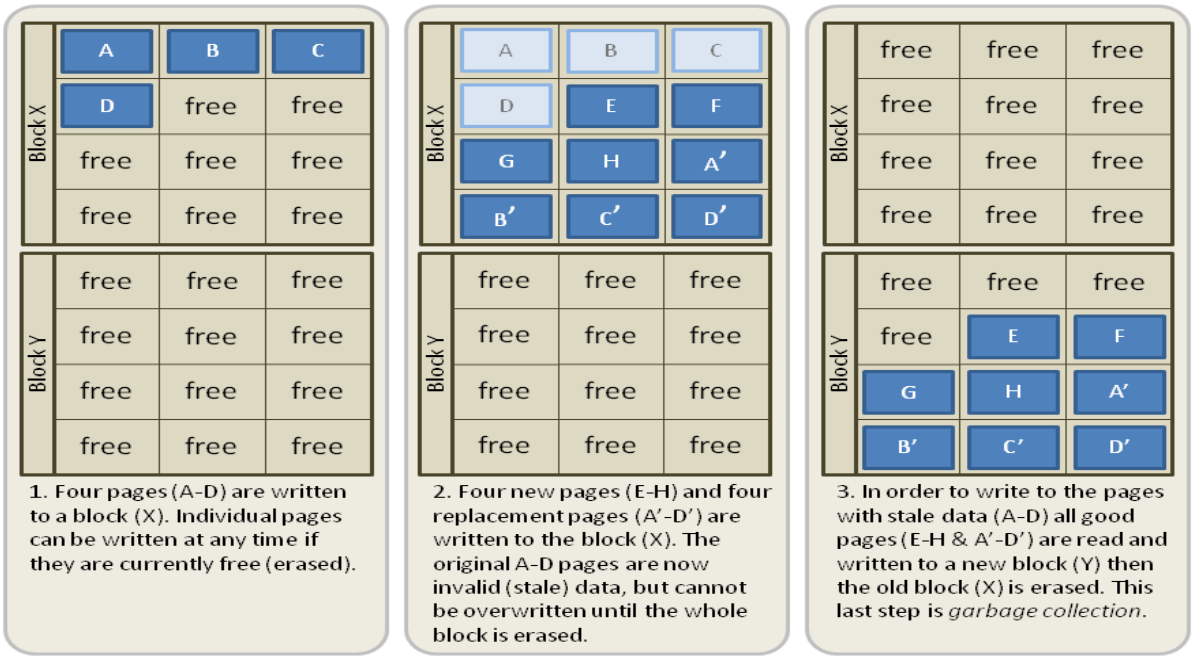
적어진다는 것을 의미한다.

SSD의 쓰기와 읽기 연산의 기본 단위는 page이지만 삭제는 block 단위로 일어나게 된다. 이것은 SSD의 물리적 특성이기 때문에 연산의 단위를 통일할 수는 없다. 또한 기존 HDD와는 다르게 write 연산 시 DISK 내에 덮어쓰기(overwrite)가 불가능하다. 따라서 데이터의 삭제 시에는 삭제되지 않아야 할 page단위의 데이터들을 다른 block으로 옮기고 block을 삭제해야 한다. 이 과정에서 삭제된 block의 상태는 invalid가 되고, 옮겨진 데이터들에 대한 mapping table을 수정해야 하는 연산이 필요하다.

삭제된 공간을 다시 사용하기 위한 garbage-collection이 필요하게 되는데, 이 과정은 디스크가 유휴 상태일 때 진행하게 된다. Invalid된 공간을 valid로 만들고 wear leveling을 위해 디스크 접근 횟수를 기록하게 된다.



<그림1. Page-FTL의 동작방식 한계 및 보완책(1)>



<그림2. Page-FTL delete연산과 garbage-collection의 예(2)>

우리는 이러한 H/W, S/W 특성에 대해 성능을 향상시킬 수 있는 방법을 생각해보게 되었고, 기존 FTL의 동작방식과 유사하지만 기존보다 성능이 향상될 수 있는 LSM-tree FTL 구조를 만들어낼 수 있을 것이라 예상한다.

2-2. 기대 효과 및 문제점

-기대 효과

기존 FTL의 역할을 수행할 수 있으며, 주제 선정 동기에서 나타난 FTL의 여러 특성에 대해 다음과 같은 장점이 있을 수 있다.

1. FTL을 LSM-tree을 이용하여 구현을 했을 경우 mapping table이 필요가 없다. 따라서 실제로 사용할 수 있는 DISK 공간이 기존에 비해 더 많아진다.

2. FTL을 LSM-tree을 이용하여 구현을 했을 경우 block단위의 수정이 한번에 이루어지므로 garbage collection이 필요하지 않게 된다(실제로는 동작 방식에 의해 garbage collection이 자동으로 이루어진다). 또한 DISK가 순차적으로(sequential) 사용되기 때문에 wear leveling이 필요하지 않게 된다.

-예상되는 문제점

LSM FTL 의 구조는 B+ tree 구조를 따르고 DISK 내부에서의 데이터는 계층을 이룬다. 계층을 이루기 때문에 DISK 내에 같은 key 값이 여러 계층에 존재할 수 있고, 이것은 읽기 연산 시에 상당한 불리함으로 작용한다. 또한 같은 특성으로 인해 여러 계층에 같은 데이터가 있을 수 있고, DISK 의 공간을 낭비하게 된다.

그러나 이러한 단점에도 프로젝트를 통해 개발될 FTL 의 종합 성능이 더 좋을 것이라 판단된다.

3. 개발 내용

3-1.개발 플렛폼

bluedbm(SSD 가상 디바이스), Linux kernel 4.x, GCC 컴파일러

3-2.상세 요구사항

-기능적 요구사항 (functional requirements)

Linux file systme에 맞는 read, write, create, delete 연산을 제공한다.

Read 연산의 경우에는 Page 단위로 이루어지며 입력으로 Logical Page Number가 들어오게 되면 이에 해당하는 Physical Page의 주소를 넘겨준다. 이렇게 넘겨받은 Physical Page의 주소를 통해서 File System이 주소에 해당하는 데이터를 읽는 연산이 이루어진다.

Write 연산의 경우에도 Page 단위로 이루어진다. 해당 데이터를 적을 Logical Page Number와 Page 단위의 데이터를 넘겨준다. 해당 데이터를 넘겨받은 후에 이를 디스크에 적어준다.

Delete 연산도 Page단위로 이루어진다. 지울 Logical Page Number를 넘겨주게 되면 해당 Logical Page Number에 해당되는 디스크 공간을 Free해준다.

Read, Write, Delete 연산 모두 기존의 FTL이 제공하는 바를 동일하게 제공해준다. 기존에 사용하는 FTL과 동일하게 지원해주어야만 상호 비교와 기존의 시스템에서 변경이 가능하기 때문에 기능적 요구사항은 동일하다. 하지만 내부적으로 구현방식과 자료구조가 상이하다.

-비기능적 요구사항 (non-functional requirements)

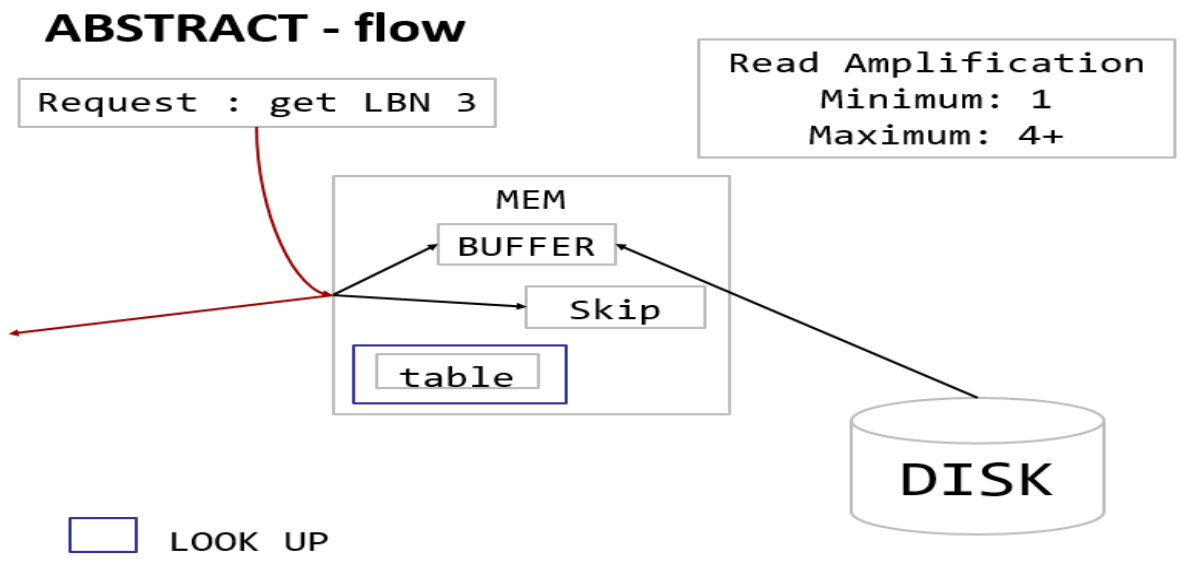
기존의 Flash Translate Layer의 비교를 통한 개선이 주요 목표인 만큼 비기능적 요구사항에서도 주로 기존의 FTL과의 차이를 통해서 서술하겠다.

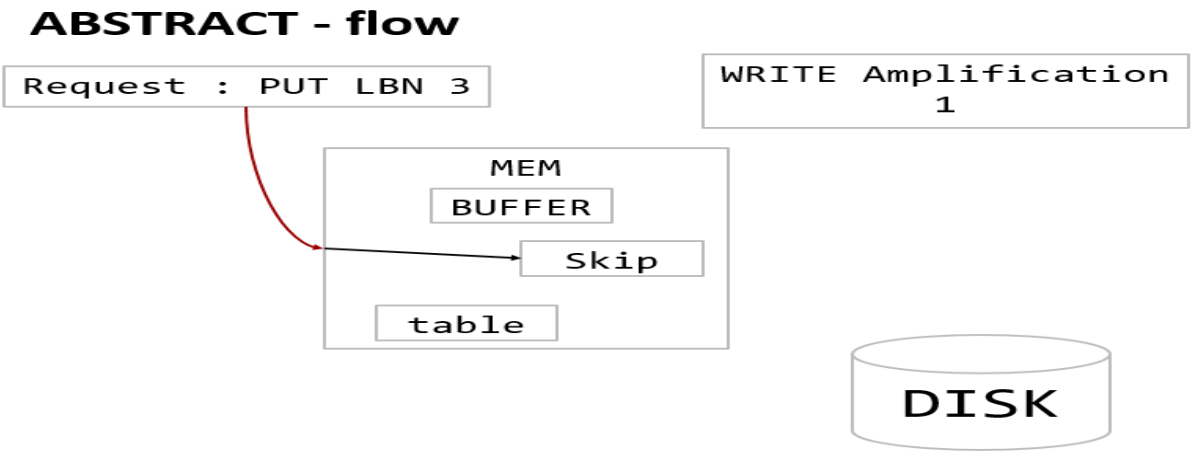
기존의 FTL과는 다르게 Page Table이 존재하지 않기 때문에 기존의 FTL에서 Page Table이 차지하는 1GB용량을 더 사용이 가능하다.

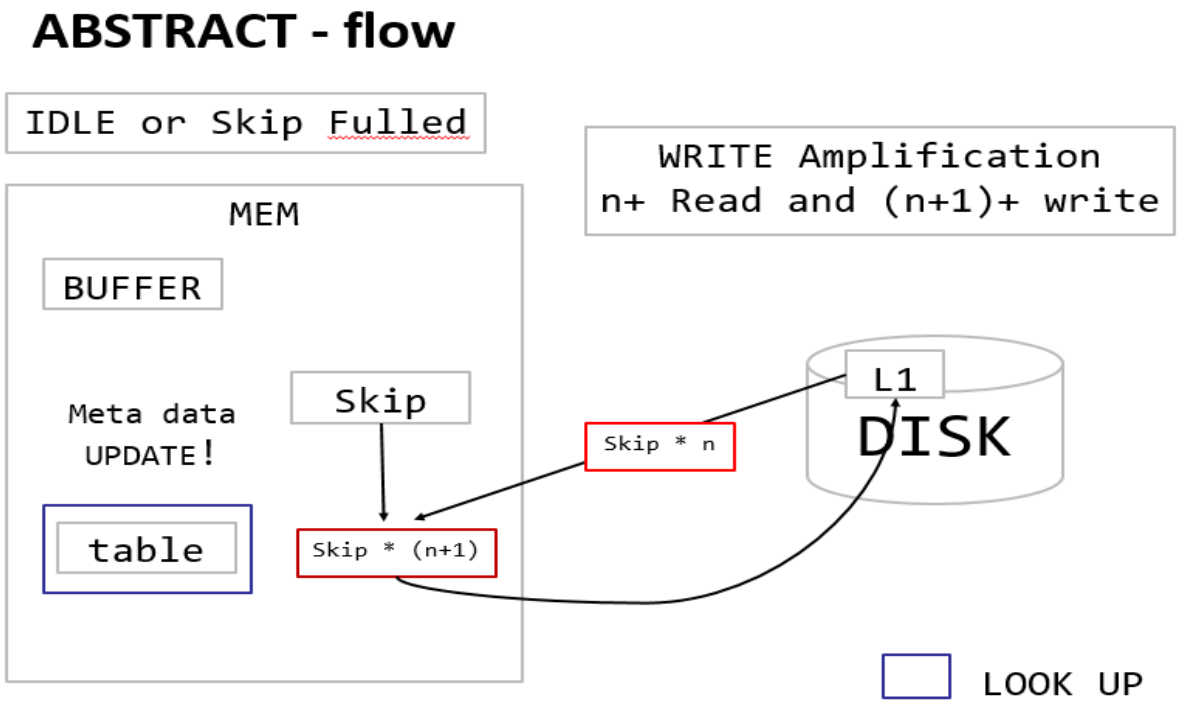
또한, 기존의 FTL에서의 Write 연산의 속도보다 LSM Tree 자료구조를 사용함으로써 이보다 더 빠른 반응속도를 지녀야 하며, 기존의 FTL에서의 Read 연산의 속도와는 비슷한 반응속도를 지녀야 한다.

Power Failure의 경우에는 개발하고자 하는 기존의 FTL과 LSM Tree 방식의 FTL의 차이 분석과는 거리감 있는 주제이므로, 발생하지 않는다고 가정하고, 기존의 FTL에서 개선하고자 하는 방향에 더 초점을 맞추었다.

-명령어 흐름







3-3. 개발자료 구조

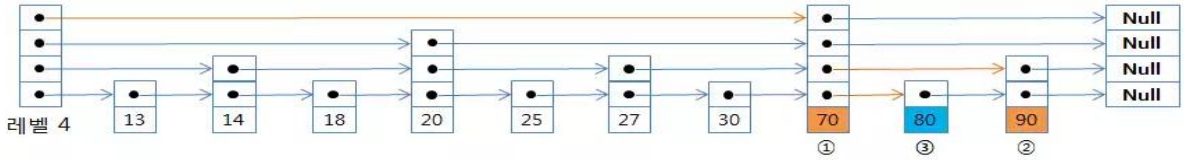
- Skip list

Skip list는 정렬된 상태를 유지하면서 데이터를 삽입, 삭제하고 탐색할 수 있는 데이터 구조체이다. 이 자료구조는 링크드 리스트의 단점을 개선하는 데서 시작한다. 여기서 설명하는 링크드 리스트는 정렬된 상태를 유지하는 리스트이다. 이런 링크드 리스트에서 n 번째 Node를 찾으려면 n 번의 비교를 해야만 한다. 즉 최악의 경우의 Search연산은 O(N) 시간이 걸린다.

이러한 탐색 연산을 줄이기 위해서 Skip list에서는 레벨을 가진다. 링크드 리스트의 Node에 여러 개의 포인터를 두어서 이 값들이 바로 다음 Node를 가르키는 것이 아닌 더 멀리 있는 Node들을 가르키게 함으로써 탐색 속도를 높이는 방법이다.

아래 보이는 그림이 Skip list의 예이다. 여기서 중간에 Node 삽입시에는 확률적으로 위의 Level의 Node들을 가지게 함으로써 아래와 같은 데이터의 형태가 유지되게 할 수 있다.

Skip list에서의 Insert, Delete, Search 연산 모두 Log(N) 시간에 수행이 가능하다.



<Skiplist 구조 (1)>

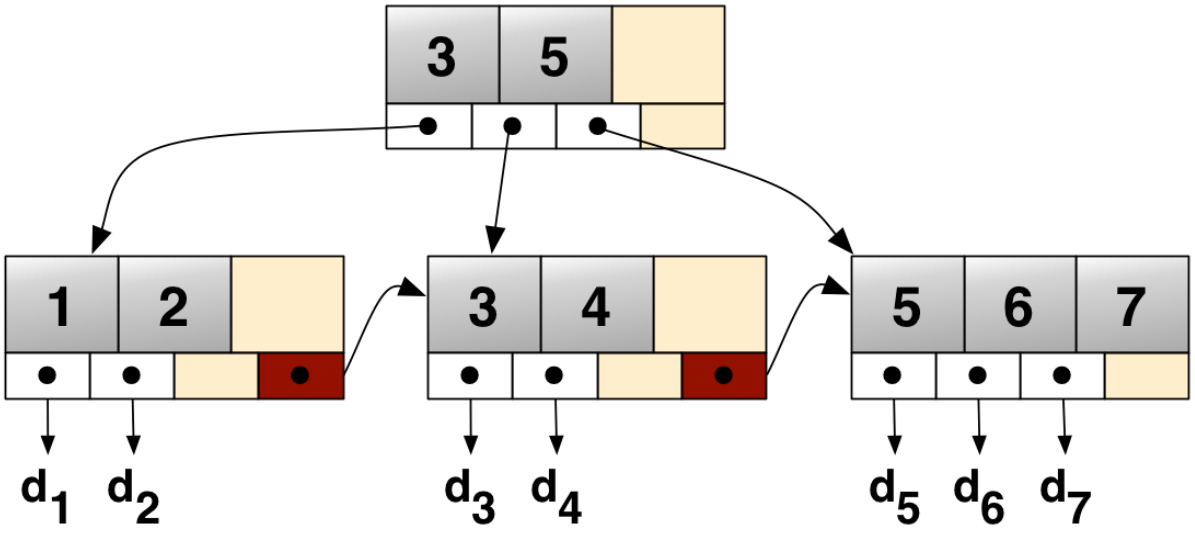
- B+ Tree

B+ Tree는 Binary Search Tree와 유사한 자료구조이다. Binary Search Tree와는 다르게 각각의 Node들의 Child가 2개가 아닌 그 이상의 Child를 가진다. 그렇기 때문에 Binary Search Tree 보다 더 적은 Level를 가지기 때문에 Search 연산을 수행하는데 있어서 유리하다. 또한, 가장 끝에 존재하는 Node인 Leaf Node들이 서로 Linked List의 형태로 연결되어져 있다.

B+ Tree에서는 Insert, Delete 연산이 수행되면서 전체 Tree를 Balance 되게 조절해주기 때문에 Worst Case의 수행시간 또한, 양호하다.

이러한 점들 때문에 B+ Tree는 검색엔진이나 DBMS와 같이 Search 연산을 많이 수행하는 저장장치에서 유리하다. 구현 목표인 FTL에서도 Search연산이 자주 발생하기 때문에 적합하다.

아래 보이는 그림이 B+ Tree의 한 예이다. 여기서 d1, d2, … , d7까지가 실제 저장되어지는 데이터들이고 나머지는 B+ Tree 형성을 위한 Entry이다.



<B+트리 구조 (2)>

- Bloom filter(3)

Bloom filter는 원소가 집합에 속하는지 여부를 검사하는데 사용되는 확률적 자료 구조이다. Bloom filter는 m비트 크기의 비트 배열 구조와 k가지의 서로 다른 해시 함수를 가지고 있다. 이러한 구조를 지니는 Bloom filter에 원소를 추가하는 경우, 추가하려는 원소에 대해 k가지의 해시 값을 계산한 다음, 각 해시 값에 대응하는 비트를 1로 설정한다. 또한, 해당 Bloom filter내에 어떤 원소를 검사하는 경우에는, 해당 원소에 대해 k가지의 해시 값을 계산한 다음, 각 해시 값에 대응하는 비트값을 읽는다. 모든 비트가 1인 경우 속한다고 판단하며, 나머지는 속하지 않는다고 판단한다.

위와 같이 해당 원소의 존재 여부를 검사하기 때문에, Bloom filter에 의해 어떤 원소가 집합에 속한다고 판단되는 경우 실제로는 원소가 집합에 속하지 않는 긍정 오류가 발생하는 것이 가능하지만, 반대로 원소가 집합에 속하지 않는 것으로 판단되었는데 실제로는 원소가 집합에 속하는 부정 오류는 절대로 발생하지 않는다.

Bloom filter의 원소 추가와 원소 검사에 걸리는 시간은 O(k)으로, 집합에 포함되어 있는 원소 수와 무관하다. Bloom filter에서 원소를 검사할 때 없는 원소가 있다고 판단될 확률은 다음과 같이 구할 수 있다. Bloom filter에 원소를 n개 추가했을 때, 특정 비트가 0일 확률은 $(1-\frac{1}{m})^{kn}$이다. 따라서 원소를 검사할 때 k개의 해시값이 모두 1이 될 확률은 $(1-\left(1-\frac{1}{m}\right)^{kn})^{k}$가 되며, 이 값은 대략적으로 $(1-e^{\frac{kn}{m}})^{k}$가 된다.

- LSM Tree(a),(b),(c)

LSM(Log-Structed Merge Tree)는 오랜 기간 동안 대량 Data를 Insert/Delete 할 때, 저비용으로 index를 제공하기 위한 disk-based Data structure이다. 그 구조로는 데이터의 Index를 저장하는 B+ Tree와 Log 역할을 수행하는 Skip list로 구성되어져 있다.

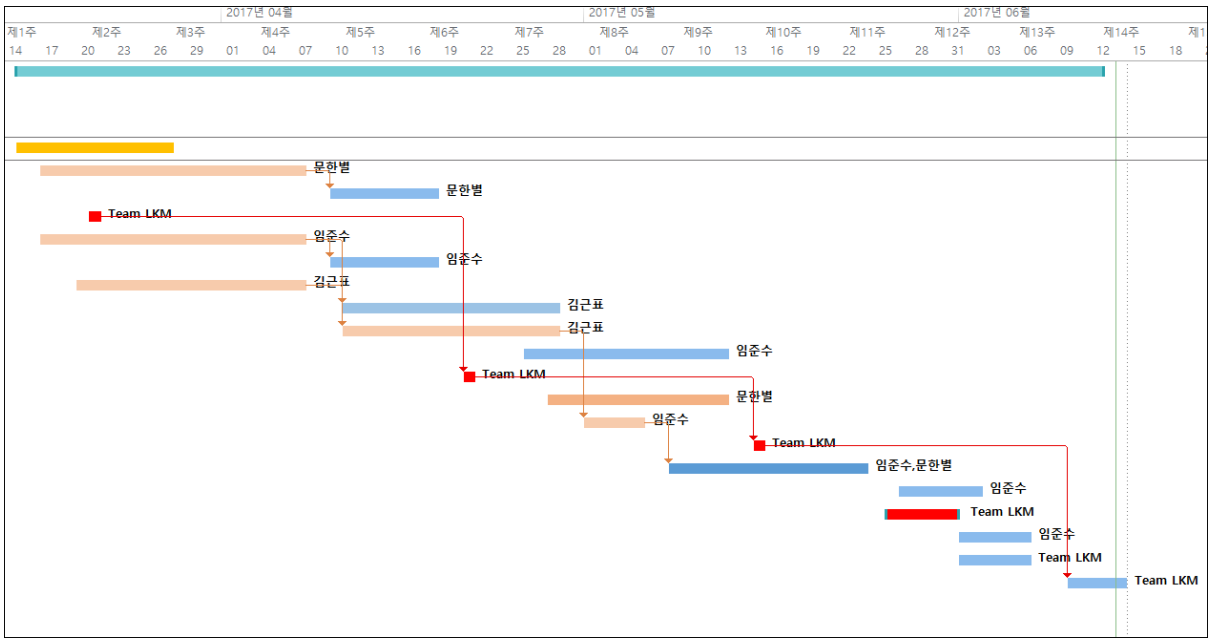
데이터를 삽입하게 되면 Log 역할을 하는 Skip list에 저장해 두었다가, Skip list가 가득 차게 되면, 이를 실제로 데이터가 저장되는 B+ tree에 내리는 형식으로 동작하게 된다. 또한, Index를 이용한 Search도 Skip list에서의 Search 이후에, 존재하면 그대로 리턴하고, 존재하지 않으면 B+ tree에서의 Search를 통해서 해당 데이터를 리턴해준다.

4. 개발 방법

4-1 간트 차트 및 역할 분담

-간트차트

작업 이름	기간	시작	완료	선행 작업	자원 이름
LSM-tree FTL을 이용한 SSD 성능 향상	66 일	17-03-15 (수)	17-06-12 (월)		
제안서 작성	9 일?	17-03-15 (수)	17-03-27 (월)		
BloomFilter 기본 개발	16 일	17-03-17 (금)	17-04-07 (금)		문한별
BloomFilter 성능 향상(해시 함수 등)	7 일	17-04-10 (월)	17-04-18 (화)	3	문한별
정기회의) 2차 회의	1 일?	17-03-21 (화)	17-03-21 (화)		Team LKM
SkipList 기본 개발과 기본 연산 구현	16 일	17-03-17 (금)	17-04-07 (금)		임준수
LSM tree를 위한 SkipList 연산 수정	7 일	17-04-10 (월)	17-04-18 (화)	6	임준수
B+ tree 개발 및 기본 연산 구현	15 일	17-03-20 (월)	17-04-07 (금)		김근표
LSM tree에 맞는 B+ tree 연산 수정	14 일	17-04-11 (화)	17-04-28 (금)	8	김근표
구현된 components를 이용한 LSM tree 개발(중복키 제거, 중간고사 기간 포함)	14 일	17-04-11 (화)	17-04-28 (금)	6,8	김근표
구현된 components를 이용한 LSM tree 개발(중복키 존재)	13 일	17-04-26 (수)	17-05-12 (금)		임준수
정기회의) 2차 회의	1 일	17-04-21 (금)	17-04-21 (금)	5	Team LKM
BlueDBM page-FTL 분석	11 일	17-04-28 (금)	17-05-12 (금)		문한별
리눅스에 맞는 LSM tree 최적화	5 일	17-05-01 (월)	17-05-05 (금)	10	임준수
정기회의) 3차 회의	1 일	17-05-15 (월)	17-05-15 (월)	12	Team LKM
LSM-tree와 page-FTL의 인터페이스 동기화	12.5 일	17-05-08 (월)	17-05-24 (수)	14	임준수,문한별
RocksDB와 FTL 인터페이스 동기화	6 일	17-05-27 (토)	17-06-02 (금)		임준수
벤치마킹 및 평가	5 일	17-05-26 (금)	17-05-31 (수)		Team LKM
추가 성능 향상을 위한 구조 변경	4 일	17-06-01 (목)	17-06-06 (화)		임준수
(additional) hlm_make_req 구현을 위한 커널코드 분석 및 수정	4 일	17-06-01 (목)	17-06-06 (화)		Team LKM
최종 보고서	4 일	17-06-10 (토)	17-06-14 (수)	15	Team LKM



<간트 차트>

-역할분담

임준수 : SKIPLIST, LSMTREE구조, LSMTREE VERSION 1,NOHOST 분석

김근표 : B+TREE,LSMTREE VERSION 2

문한별 : BLOOMFILTER, 가상화 Blue DBM 분석

4-2 개발 방법 상세

-B+ tree 구현(김근표)

LSM Tree의 구성요소 중의 하나인 B+ tree를 구현하였다. LSM Tree에서 저장해야 하는 데이터가 (Logical Page Number, Physical Page Number)의 데이터 pair이다. 따라서 B+ tree에서의 Index는 LPN이 되고, 해당되는 데이터는 PPN이 된다.

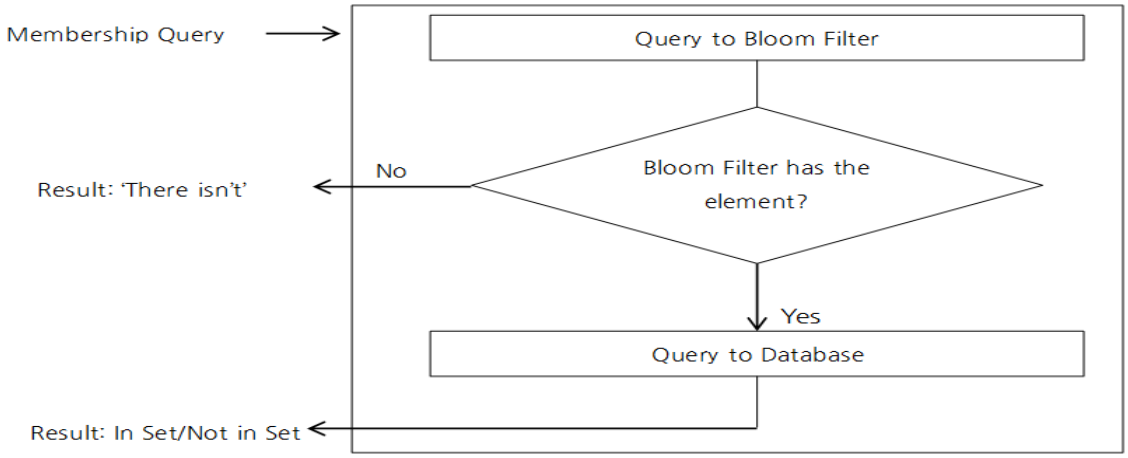
또한, B+ Tree의 끝에 있는 Node인 Leaf Node에는 위에서 언급했던 Bloom Filter를 넣어두었다. 이를 통해서 LSM Tree에서 B+ Tree의 Node에 대한 접근을 하고, 해당 데이터가 실제로 있는지를 미리 Bloom Filter를 통해서 빠르게 확인할 수 있다.

해당 자료구조를 구현하는 구현 환경 자체가 메모리에 민감한 환경이므로 재귀적인 함수 호출이 아닌 반복문만을 이용하여 구현하였다.

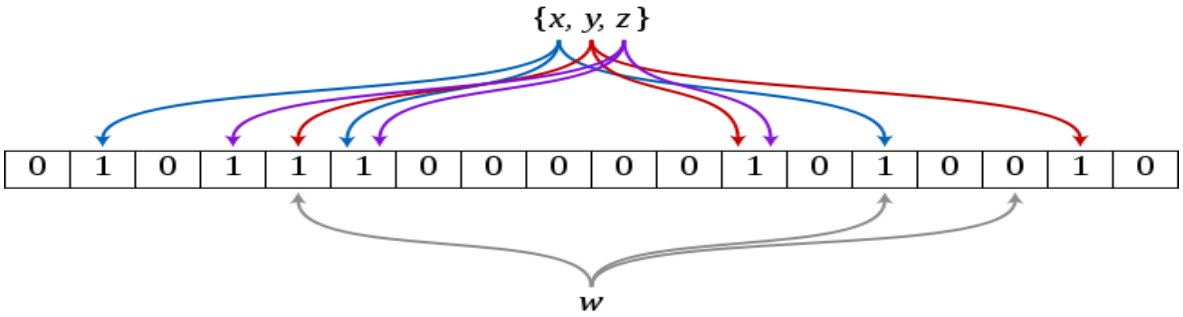
해당 자료구조 접근의 편리성을 위해서 Iterator를 구현하였다. Iterator을 통해서 search시에 반환하고 여기서 다음 Node로의 접근이 가능하기 때문에 내부 구조에 대해서 모르더라도 편리하게 자료구조를 이용할 수 있다.

-Bloom filter(문한별)

Bloomfilter 는 확률적 자료구조로 어떤 원소가 어떤 집합의 원소인가를 확률적으로 판단하는 자료구조이다. 즉, 어떤 원소 x 와 집합 A 대해 bloomfilter 를 이용한 결과가 '참'이라고 해도 반드시 x 가 집합 A 에 포함되는 것은 아니다(false positive). 그러나 실제로 데이터가 있는데 없다고 잘못 판단하는 경우(false negative)는 없다. 즉, bloomfilter 의 결과는 '아마도 존재한다.' 와 '확실히 존재하지 않는다.' 의 두 가지이다. 여러 개의 해시 함수를 이용하여 값에 대한 bit 를 1 로 만든다.



<그림 1. Bloomfilter 의 특성 flowchart(1)>



<그림 2. Bloomfilter 예제(2)>

위 그림 2 의 예제를 보면, 원소 x, y, z 에 대해 각각 3 개의 해시 함수를 사용한 값의 bloomfilter 비트 배열을 1 로 만드는 모습이다. 이 때, 원소 w 가 존재하는 지에 대한 질의(query)를 하게 되면 그 값은(1,1,0)으로, 원소 w 는 확실히 존재하지 않는다는 것을 알 수 있다.

그러나 만약 원소 a 에 대한 질의에서 (1,1,1)의 bloomfilter 값을 얻었다면, 실제로는 원소 a 가 존재하지 않지만 존재한다고 잘못 말할 수 있다.
즉, bloomfilter 값이 하나라도 0 이 나오면 그 원소는 확실히 존재하지 않지만, 모두가 1 이 나온다고 해서 그 원소가 반드시 존재하는 것은 아니다.

우리는 이러한 bloomfilter 의 특성을 이용해서 2-2 의 예상되는 문제점을 효과적으로 없앨 수 있다. DISK 내부가 계층을 이루므로 같은 key 값이 중복되어 있을 수 있는데, key 값에 대한 질의에 대해 bloomfilter 를 이용하면 '확실하게 존재하지 않는' 경우를 건너 뛸 수 있기 때문이다.

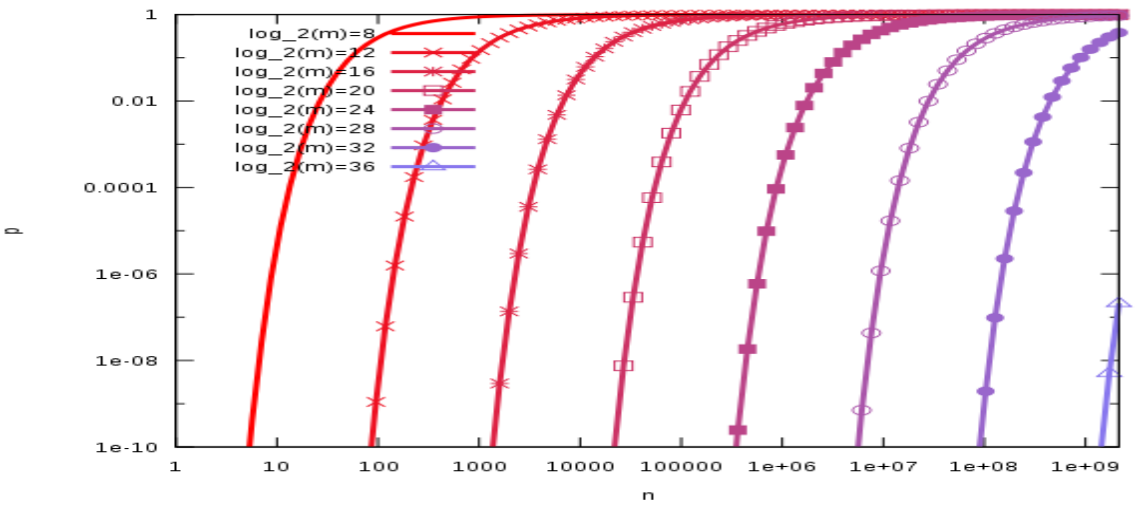
그러나 '아마도 있는' 경우에는 추가 질의가 필요할 수 있다. 있다고 했을 때 확실하게 있는지는 모르기 때문이다. 따라서 이 경우가 최소가 될 수 있는 방법을 찾아야 하는 것이 중요하다. 얼만큼의 확률을 만들 것인가를 정한다면 데이터의 개수와 bloomfilter 의 해시 함수의 개수, bloomfiler 의 비트 배열 크기에 대한 수식을 이용해 적절한 해시 함수의 개수를 구할 수 있다.

수식으로 확인하지 않아도 직관적으로도 알 수 있겠지만, false positive 가 일어날 확률은 비트 배열의 크기인 m 이 클수록, 원소 개수인 n 이 작을수록 낮다. 해시 함수 개수 k 의 경우에는 일정 수준으로 증가할 때까지는 false positive 가 일어날 확률이 낮아지지만, 일정 수준 이상 넘어서면 false positive 가 일어날 확률이 높아진다. 게다가 해시 함수 수행 비용 또한 고려해야 하기 때문에 연산 비용을 고려한 k 값을 선택해야 한다.

최적의 k 값은

$$k = \frac{m}{n}\ln 2, \quad \left(\frac{1}{2}\right)^k = \frac{1}{2}^{\frac{m}{n}\ln 2} = \left(\frac{1}{2}^{\ln 2}\right)^{\frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

의 수식으로 나타낼 수 있으며, 이 경우에 대한 false positive 발생 확률 p 는 다음과 같다.



<그림 3. 함수 개수가 최적일 때, m 과 n 에 따른 false positive 확률(3)>

LSM-FTL 은 각 block 당 key 값이 1000 개가 필요하고, 비트 배열의 크기는 정할 수 있기 때문에 $\left(\frac{1}{2}\right)^k$ 이 원하는 만큼 작아지는 수 k 를 정하면 된다. $k = \frac{m}{n}\ln 2$ 이고 n = 1000 이므로, 비트 배열의 크기 m = 8192 라 하면, k 는 4.xxx 의 수가 되고, 개수를 4 개로 했을 시 약 7%의 false negative 을 가지게 된다.

위 수식의 결과를 이용해서 4 개의 해시 함수를 가지도록 bloomfilter 를 구현하면 예상되는 문제점을 최소화시킬 수 있다. key 값을 한 번 접근 시 원하는 데이터가 아니라면 바로 다음 계층으로 가서 찾으면 되고, key 값이 존재하지 않는다고 하면 DISK 접근 없이 다음으로 가서 찾을 수 있기 때문이다.

따라서 visual studio 를 이용해 bloomfilter 를 개발하는 작업을 하였으며, 해시 함수는 division method 를 이용했다. 2^n에 가까운 소수를 이용해 각 키 값에 대한 충돌의 가능성을 개발 조건 내에서 최소화했다. Read 연산 시 데이터를 찾기 위한 key 값을 입력 받으면 bloomfilter 를 통해 찾는 데이터에 대한 접근시간을 비약적으로 줄일 수 있다.

-Skiplist 구현(임준수)

Skiplist의 장점은 restructuring을 굳이 하지 않아도 확률적인 구조로 binary search tree의 형태를 유지할 수 있다는 점이다. Skiplist는 각각의 노드가 개별적인 level을 가지고 있고 이 level을 통해서 탐색을 효율적으로 진행할 수 있다. 이 때 level을 정하는 것은 확률적인 구조를 따르고 있는데 예를 들어 1/2확률로 level을 증가한다면 동전 던지기와 같이 앞면이 나올 때까지 동전을 던지고 던진 횟수로 level을 결정을 하는 것이다. Level을 정하는 확률은 낮을수록 노드 당 레벨은 낮아 지고 이는 검색의 효율성을 떨어뜨리지만 메모리를 적게 사용하는 경향이 있다.

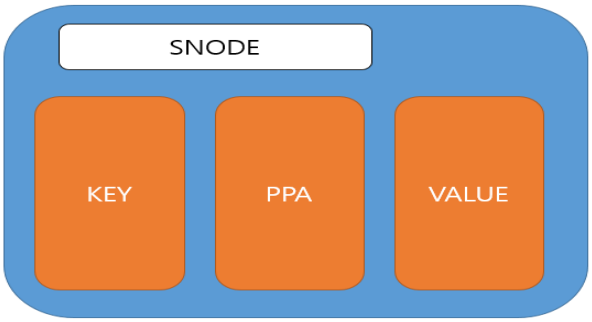
다음은 skiplist의 level 확률에 따라 탐색시간과 memory 노드당 평균 level을 보여주는 표이다.

확률	탐색 시간	노드당 평균 레벨
1/2	1	2
1/e	0.94	1.58
1/4	1	1.33
1/8	1.33..	1.14
1/16	2	1.07

<Skiplist 확률에 따른 여러 변수들(4)>

탐색시간과 메모리효율(노드당 평균 level)을 고려했을 때 1/4확률일 때 가장 효율적으로 판단되었기 때문에 제작한 프로그램에서의 확률은 1/4로 만들었다.

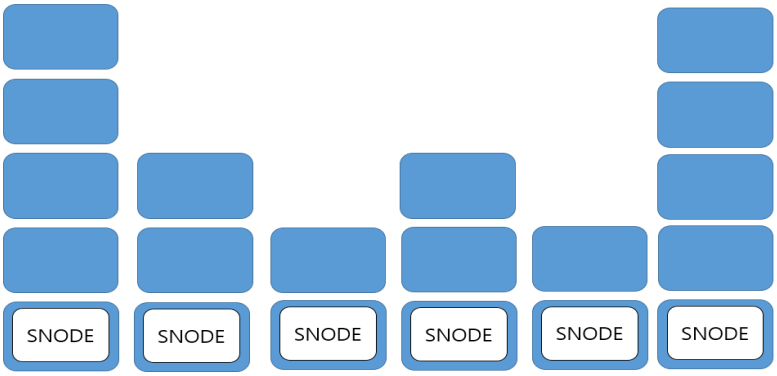
다음은 skiplist를 구성하는 노드인 SNODE의 모습이다.



<Skiplist SNODE 구조 모습>

KEY는 사용자의 입력으로 들어오게될 LPN(logical page number)이고 PPA(Phsical page number)는 실제 SSD 안에 적혀질 위치를 가지게 된다. VALUE는 그 때의 데이터 값을 의미한다. PPA가 존재하는 이유는 skiplist를 저장하게 될 때 meta data와 data들을 구분하기 때문에 그 위치를 저장해야 하기 때문이다.

다음은 skiplist의 대략적인 모습을 나타낸 그림이다.

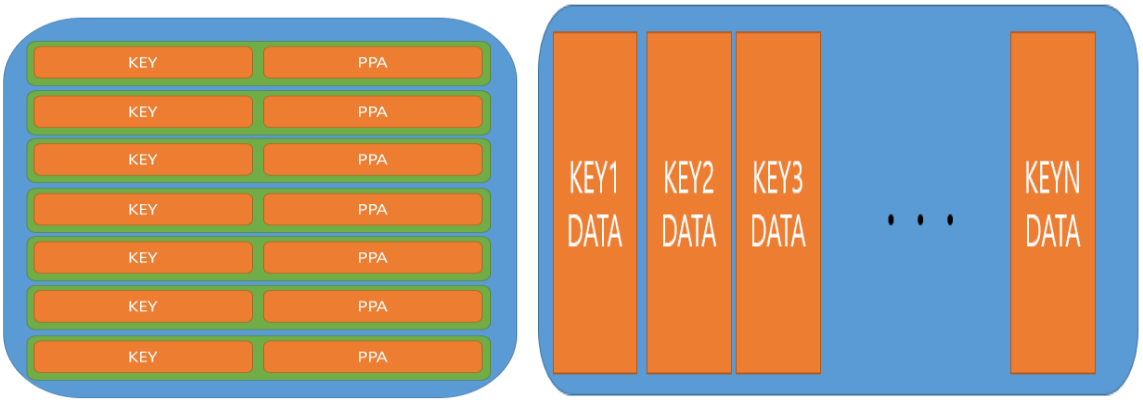


<Skiplist 구조 모습>

Skiplist의 문제점 중 하나는 바로 실제 데이터를 읽을 때 이다. 만약 skiplist의 구조를 그대로 저장소에 적게 된다면 그 데이터를 읽고 그 데이터들을 이용해 노드를 검색하여야 한다. 이는 skiplist에 대한 데이터를 저장소에서 읽는 시간을 제외하고도 skiplist를 구성하는 시간(O(NlogN))과 검색하는 시간(O(NlogN))이 걸리기 때문에 매우 비효율적이게 된다. 그래서 데이터를 저장할 때 skiplist의 구조대로 저장하지 않고 skiplist의 level 1에서는 모든 노드들이 정렬된 상태로 존재한 다는 것을 이용해 table 형태로 저장을 하였다. 이렇게 되면 데이터를 읽을 때 skiplist를 구성할 필요 없이 검색만을 하면 되기 때문에 binary search의 비용(O(logN))만 있으면 된다. 이로 인한 단점은 읽은 데이터를 skiplist로서 활용하지 못한다는 점인데 LSM-TREE 특징인 새로운 데이터가 삽입될 시에는 새로운 공간에 적히는(기존의 데이터가 바뀌지 않는) 점을 이용하기 때문에 굳이 기존 데이터를 skiplist로 만들 필요가 없다.

다음 왼쪽 그림은 skiplist가 저장될 때 변환되는 table인 keyset의 대략적인 모습이다.

오른 쪽 그림은 meta data(keyset)와 다른 공간에 적혀져 있는 DATA들이다. 이렇게 meta data와 data들을 분리 시킴으로써 LSM TREE의 COMPACTION 연산을 진행할 때 meta data만을 가지고 진행 할 수 있게 된다.



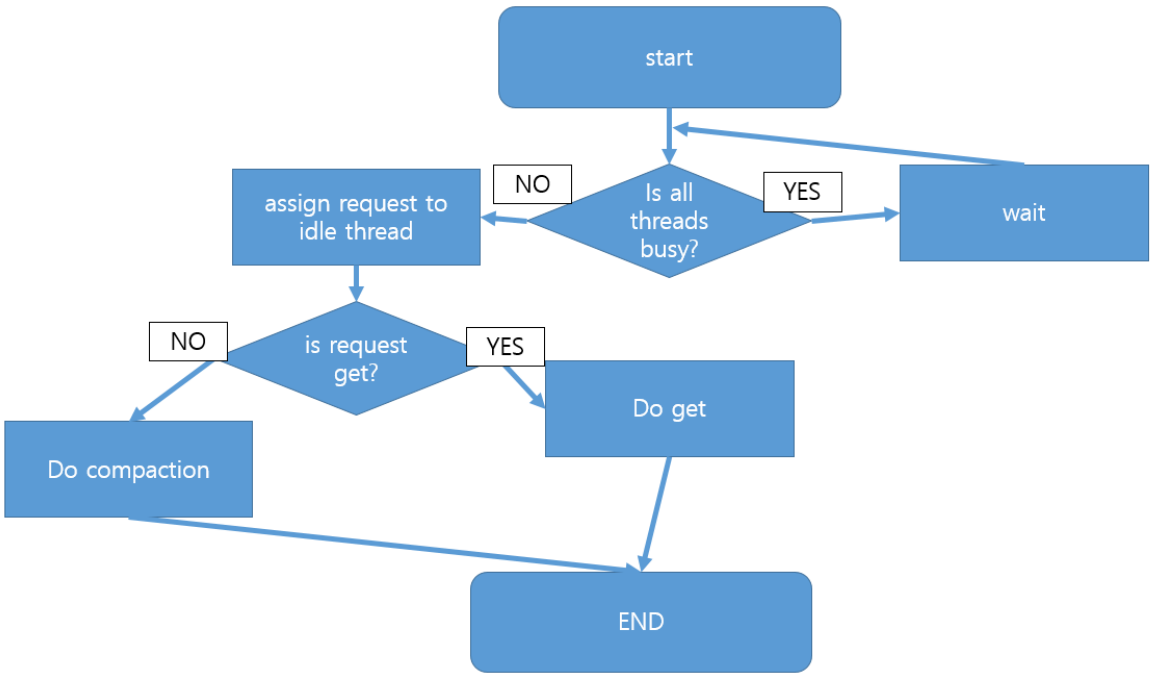
<Skiplist가 저장될 때 그림 좌:METADATA 우:DATA>

-LSM tree 구현:version2 (임준수)

LSM-TREE의 주된 연산은 사용자로부터 key(LPN)와 value를 받아서 데이터를 저장하는 put 연산과 key를 받아서 해당 데이터를 주게 되는 get 연산, 그리고 각각의 레벨이 가득 차게 되었을 때 다음 레벨로 데이터들을 옮겨 주게 되는 compaction(또는 merge) 연산이 있다. 이때의 compaction 연산에 따라서 LSM-TREE의 버전이 달라지게 된다.

우선 put 연산을 보게 된다면 level0인 skiplist가 가득 찼는지 판단하고 가득 찬 상태가 아니라면 주어진 key와 value를 입력하게 연산을 종료하고 가득 찬 상태라면 compaction 연산을 호출한다.

Compacion 연산과 get 연산은 모두 Multi-thread를 이용해 진행 되는데 multi-thread는 thread-pool을 이용해서 구성되어 있다. 다음은 thread-pool의 flow chart 이다.

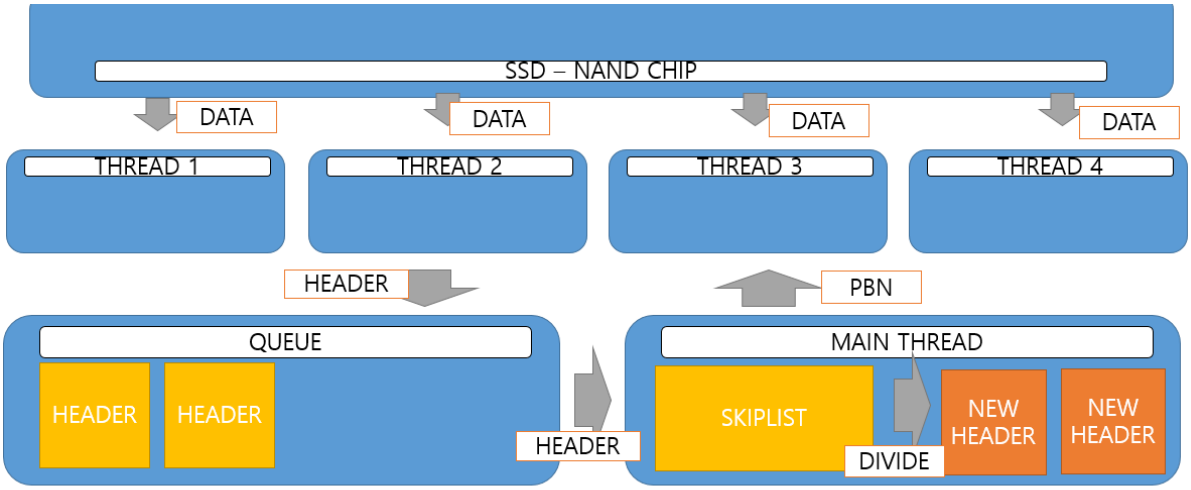


<THREAD POOL flow chart>

LSM TREE는 언급했다시피 compaction 방법에 따라 2가지 버전을 나뉠 수 있다. 중복을 허용하는 버전과 중복을 허용하지 않는 버전이 있는데 중복을 허용하는 버전(version 2)(b)은 다음과 같은 형식으로 동작한다.

Compaction이 일어나면 Level 1부터 차례대로 Level이 가득 차있는지 판단하고 가득 차지 않은 Level이 될 때까지 내려간다. 그때의 가득 차지 않은 Level을 N+1이라고 하자 그러면 Level N에서부터 가장 오래된 4개의 ENTRY를 뽑아 내어 중복을 없애 주는 연산을 진행한다. 이때 Thread를 통해 4개의 Entry의 각각의 header를 읽게 한다. 이는 producer – consumer 형태로 4개의 producer thread가 읽기를 완료하여 지정된 큐에 헤더 데이터를 집어 넣고 consumer thread가 큐에 들어있는 thread를 읽어 하나의 skiplist를 구현 한다. Skiplist는 지정된 개수만큼 분할 되어 N+1로 내려가게 된다. N->N+1로의 작업이 끝났기 때문에 N-1->N의 작업을 수행하여야 하는데 유저레벨의 프로그램이 아니기 때문에 재귀적인 호출은 좋지 않아 do~while의 형태로 바꾸어서 N이 1이 될 때까지 진행한다.

다음은 producer – consumer 형태로 진행되는 version2를 그림으로 나타낸 것이다.



<VERSION 2 WITH THREAD>

생성된 NEW HEADER들은 다시 SSD NAND CHIP에 적히게 된다.

-LSM Tree Version 1(김근표)(c)

구현 하였던 LSM Tree의 방식 중에 첫 번째 버전을 구현하였다.

LSM Tree를 구성하는데 있어서 Skip list를 Level 0라고 하고 B+ Tree도 여러 Level로 구성되어져 있을 때, 높은 Level에서 낮은 Level로 데이터를 옮겨주어야 한다.

그 방식이 원래 LSM Tree의 동작 방식인 Merge연산이다. 이 연산을 통하면 높은 Level에 존재하는 데이터와 낮은 Level에 존재하는 데이터를 통합함으로써 두 개의 Level간의 데이터의 중복을 없애주고 이를 낮은 Level에 다시 적어주는 방식이다.

이러한 방식을 사용하게 되면, 데이터의 중복이 전혀 존재하지 않는다. 따라서 Search 연산을 수행하는데 있어서 빠르게 수행이 가능하게 된다. 하지만, 이러한 Merge 연산의 경우에는 최악의 경우에는 낮은 Level에 존재하는 데이터를 모두 메모리에 올리고 해당 데이터를 변경해주어야만 하기 때문에 Write연산의 속도가 느리다.

또한, 이 방식으로 구현하게 되면 Bloom Filter가 필요 없게 된다. 위에서 언급한 바와 같이 하나의 Level에 해당하는 B+ Tree에서는 데이터가 중복이 발생하지 않는다. 따라서, B+ Tree의 Key값에 해당하는 데이터를 Search하는데 있어서도 바로 접근이 가능하다. 굳이 Bloom filter를 통해서 그 데이터의 값이 실제 존재하는지 확인할 필요가 없다.

이와 같은 방식 또한 producer-consumer 모델과 큰 차이점이 없기 때문에 앞서 구현한 THREADPOOL을 동일하게 이용한다.

LSM TREE FTL은 여러 가지 변수가 존재할 수 있는데 THREAD의 개수, LEVEL의 개수 SKIPLIST에

담을 수 있는 KEY에 개수, PAGE SIZE 등등 configuration할 수 있는 여지가 많다.

그렇기 때문에 LSM TREE FTL 컴파일 타임에 이러한 변수들을 메크로로 설정을 해두어 re-configuration 가능하게 만들어 주었다

-BLUE DBM 분석(문한별)(d)

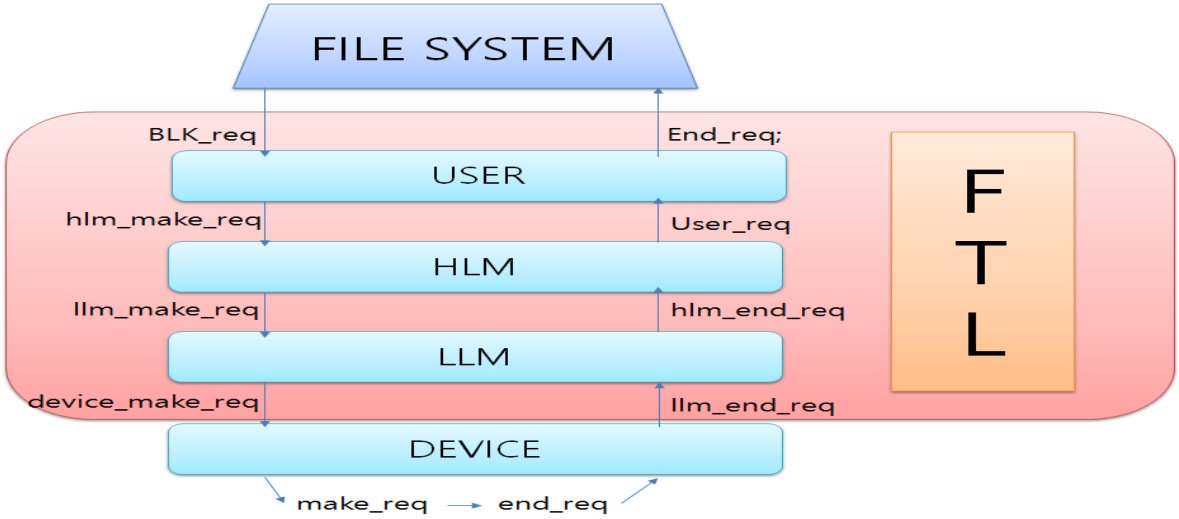
BLUE DBM 을 통해 page-FTL 과 LSM-FTL 의 성능을 비교해야 하기 때문에 기존에 BLUE DBM 내의 page-FTL 에 대한 연산 및 내부 구조를 lsm-FTL 의 구조에 맞춰야 한다.
이 과정을 위해 내부 page-FTL 을 분석해야 하는데, page-FTL 는 page-FTL 생성, get_free_ppa(write 연산), map_lpa_to_ppa(read 연산), need_gc(garbage collection 여부 파악), do_gc(garbage collection 실행) 등으로 구성되어 있다. 이 구성 함수들을 우리가 사용할 LSM-FTL 에 맞게 고쳐야 한다.

우선, page-FTL 함수의 하는 일은 사용 가능한 block 을 구하고, mapping table 을 생성하고, garbage collection 을 위한 공간을 설정한다. mapping table 의 경우는 필요가 없다. garbage collection 의 경우도 LSM-FTL 은 자동으로 이루어지기 때문에 사용이 가능한 block 에 대한 정보를 얻는 함수로만 변환하면 된다.

map_lpa_to_ppa 함수는 내부적으로 get_ppa 함수 호출을 통해 read 연산을 진행하게 된다. read 연산 시 메모리에 올리는 것은 인터페이스를 맞추기 어렵지 않다. 내부 구조만 바뀌었을 뿐, 연산 시 수행하는 동작은 같기 때문이다.

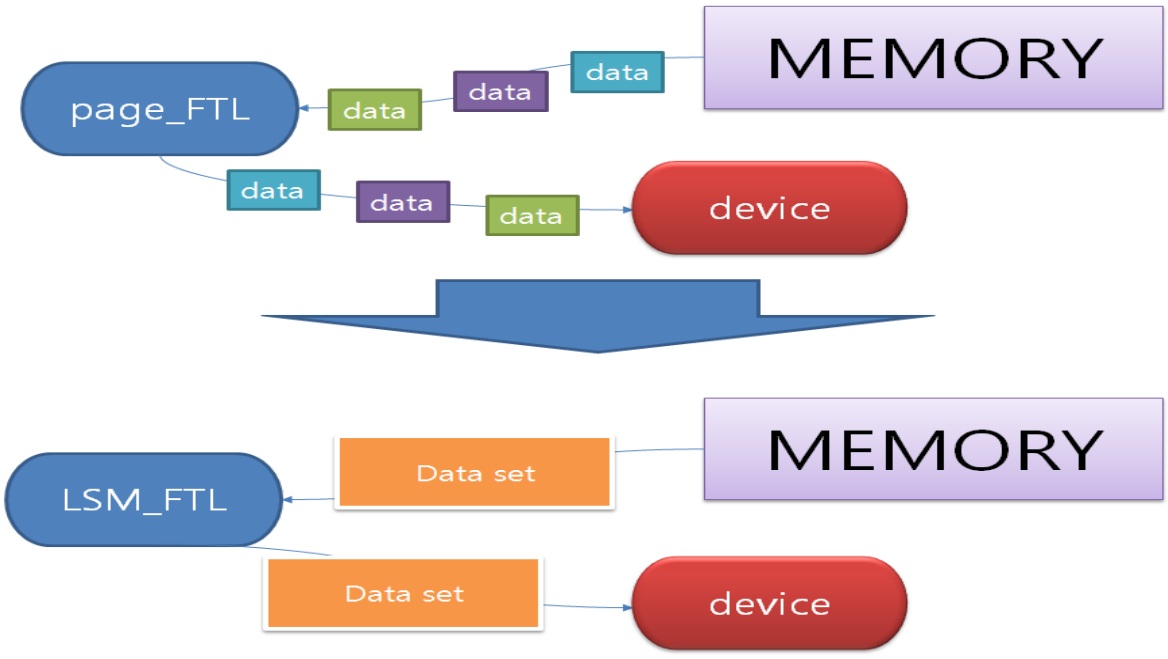
need_gc 함수와 do_gc 함수 역시 garbage collection 시 사용되는 함수인데, LSM-FTL 의 경우에는 삭제연산 시 garbage-collection 이 발생하지 않으므로 구성을 바꾸기가 어렵지 않다. need_gc 함수 내부에서는 사용 가능한 block 이 전체의 일정 비율 이내인지를 확인하고 do_gc 함수 내부는 사용 가능한 block 을 만들기 위해 list 구조로 묶인 invalid block 을 valid 하게 바꾼다.

남은 함수는 get_free_ppa, write 연산을 담당하는 함수이다.



< FTL 의 data flowchart>

위 그림을 보면 데이터가 device 에 저장되기까지의 흐름을 알 수 있다. write 연산 시, 각 단계를 거쳐 최종적으로 user→file system 으로 end_request 를 보내서 write 연산이 끝났다는 것을 알 수 있다. 그러나 LSM-FTL 의 경우 write 된 데이터를 메모리에 올려놓고, block 단위의 데이터가 전부 쌓였을 때 device 로 BLK_request 를 보내게 된다. 즉, file system 에서는 write 연산을 실행했는데 end_request 를 받지 못하는 상황이 발생한다.



<page FTL 과 LSM FTL write 연산 비교>

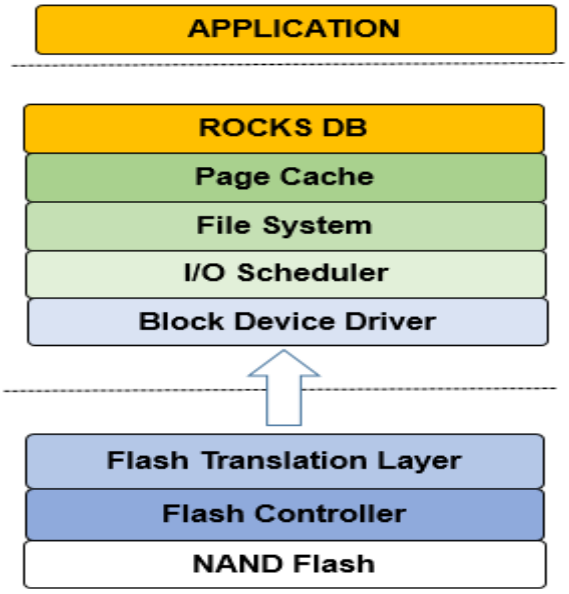
이런 상황을 해결하기 위해서 수정해야 할 것은 get_free_ppa 함수 뿐 아니라 kernel 의 함수까지 분석하고 수정을 해야 한다. 그러나 kernel 내부의 수정은 시스템 전체에 큰 영향을 미칠 수 있기

때문에 신중히 시간을 들여서 분석을 해야 한다. 시간의 제약이 있는 이번 경우에는 kernel 함수 분석은 현실적으로 불가능하다고 보아야 할 듯 하다.

-NOHOST 분석(임준수)

LSM TREE FTL을 올리기 위한 가상화 플랫폼인 BludeDBM에서의 어려움이 발견되었기 때문에 개발 방향을 바꿀 수 밖에 없었다. 주된 목표는 LSMTREE를 이용해 기존의 FTL(Page FTL)보다 성능이 좋게 만드는 것이었기 때문에 새로운 플랫폼에서 개발을 진행하도록 하였다.

빅데이터와 AI가 트렌드가 되고 있는 시점에서 기업에서는 좀 더 효율적인 물리적 공간에서 저전력의 시스템을 사용하기를 원하고 있다. 이러한 저전력 시스템(ARM CPU)은 상대적으로 x86 CPU기반의 서버보다 성능이 떨어지게 된다.

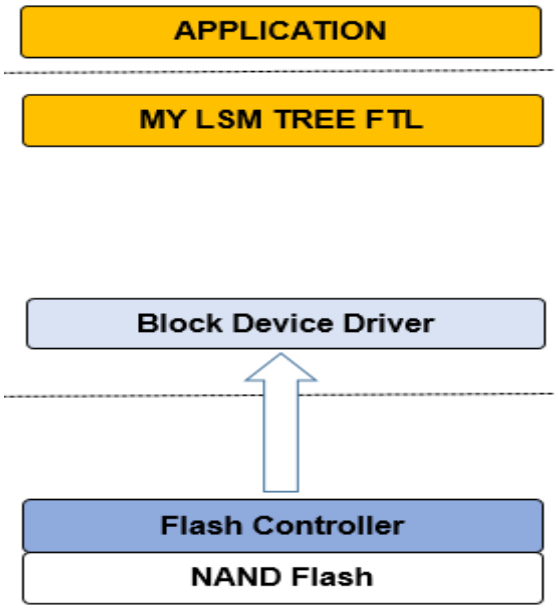


<NOHOST system 에서의 S/W 계층 구조 >

위의 소프트웨어 계층을 보자면 맨 위의 계층에는 APPLICATION계층으로 client의 요청을 처리하는 여러 가지 응용 프로그램들이 돌아간다. 그 아래에는 Rocks DB 계층은 빅 데이터를 처리하는 LSM tree 구조로 이루어진 시스템이다. RocksDB 계층을 거쳐 File system과 Driver 계층에서의 processing 후 실제 SSD 디바이스로 데이터가 보내지게 된다. 이때 SSD 내부에 존재하는 Flash Translate Layer(FTL)계층 HDD 위주의 File System을 SSD 구조에 맞게 데이터를 변환시켜주는 Layer인데 이 때 SSD구조가 LSM Tree의 동작에 유리하기 때문에 FTL의 변환 동작은 LSM TREE와 유사하다. (사용되는 FTL은 PAGE FTL이다)

비록 완성된 LSM TREE FTL이 실제 가상 디바이스에 올려 테스팅을 하는 것은 실패하였지만 Rocks DB 계층이 LSM TREE로 구성되어 있고 FTL동작 또한 LSM Tree의 동작이 유사하기 이러한

시스템에서의 완성된 LSM TREE FTL의 활용하여 해결해 저 성능의 CPU 시스템에서도 효율적으로 돌아가는 S/W를 만들 수 있겠다는 생각이 들었다.

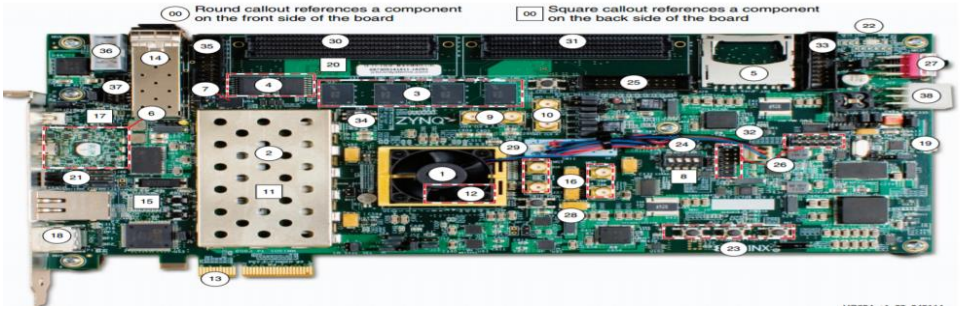


<NOHOST system 올릴 새로운 S/W 계층 구조 >

위와 같은 구조에서는 FTL 계층이 유저레벨로 올라와 있기 때문에 제작한 LSMTREE FTL또한 유저레벨에서 동작하도록 하게 구현하면 되기에 커널 함수를 사용해야 되는 부담이 없어졌다.

RocksDB에서 FTL로 넘어오는 부분은 단순한 Request 형태인데 Request type과 LPN 그리고 value 정도만이 넘어오기 때문에 완성된 LSMTREE-FTL에서 크게 손 볼 곳은 없었다. 하지만 SSD에 접근하는 것은 새로운 라이브러리인 libmemio라는 라이브러리를 이용해야만 접근이 가능했고 단순히 system call인 write 함수 대신 memio_write로만 형식을 바꾸면 되었다.

이 시스템이 돌아갈 장치의 spec은 다음과 같다. RAM : 1G , CPU :0.8MHz, SSD :16GB (SCU 706 board)

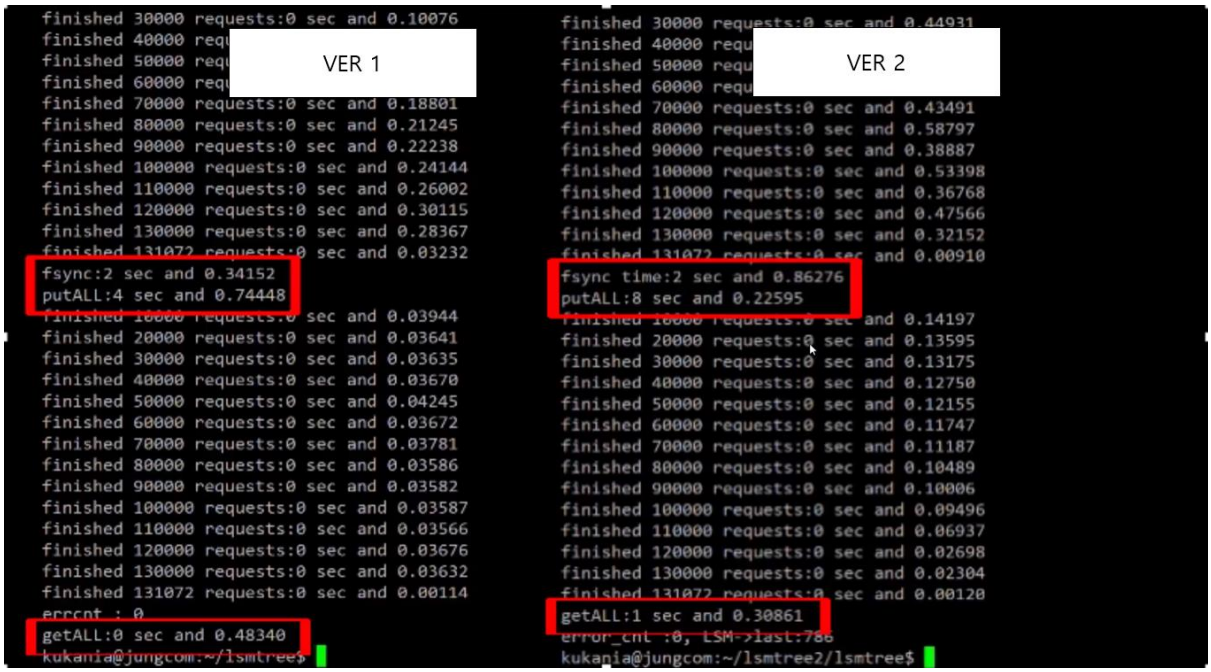


<SCU 보드 그림(e)>

5. 개발 결과

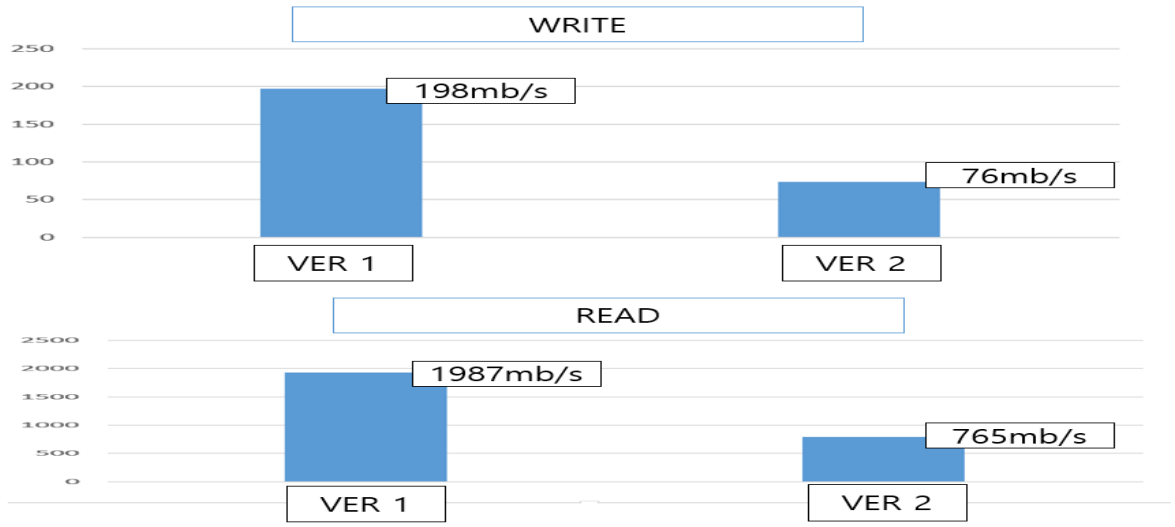
5-1 버전 비교

다음은 두 가지 LSM TREE FTL version에 대해 x86 CPU에서의 비교를 실행하는 캡처와 결과를 보여주는 차트이다.

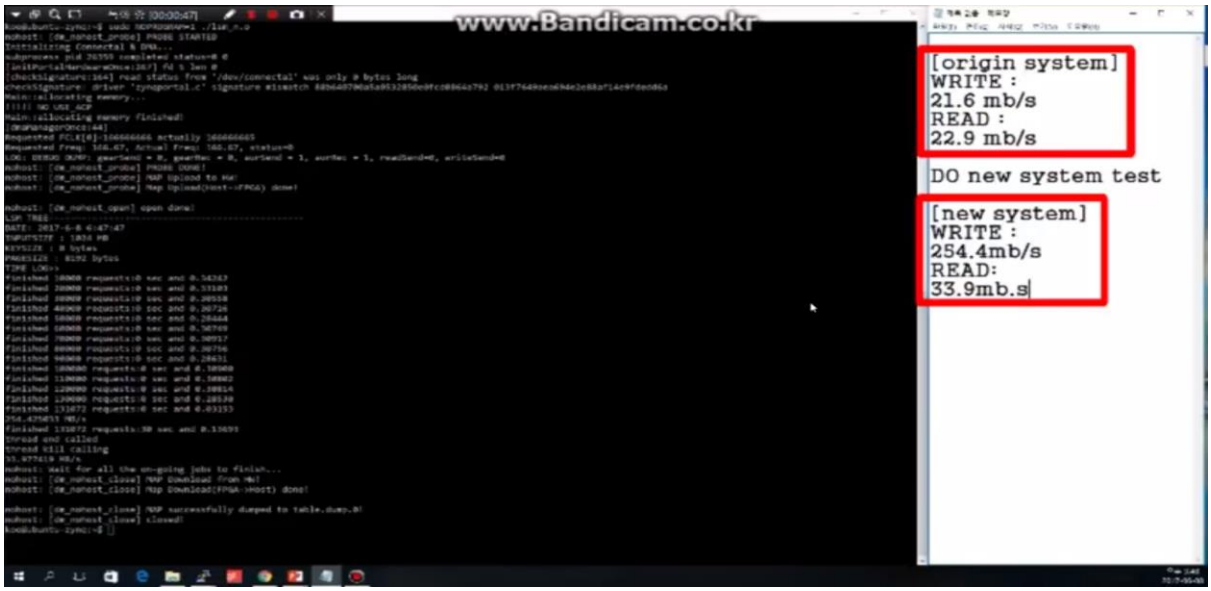


테스팅 환경이 단순한 컴퓨터이기 때문에 파일 시스템에서의 버터링이 존재하게 된다. 이 때 버퍼링 시간은 fsync time으로 나타나는 시간들이다. 그 다음 줄에 나오는 putALL이 실제 데이터를 모두 처리하는데 걸리는 시간이다. 즉 putALL – fsync를 한 결과가 실제로 CPU가 사용되는 시간이다. CPU가 시간이나 모든 request를 처리하는 시간이 월등히 version 1이 뛰어났다.

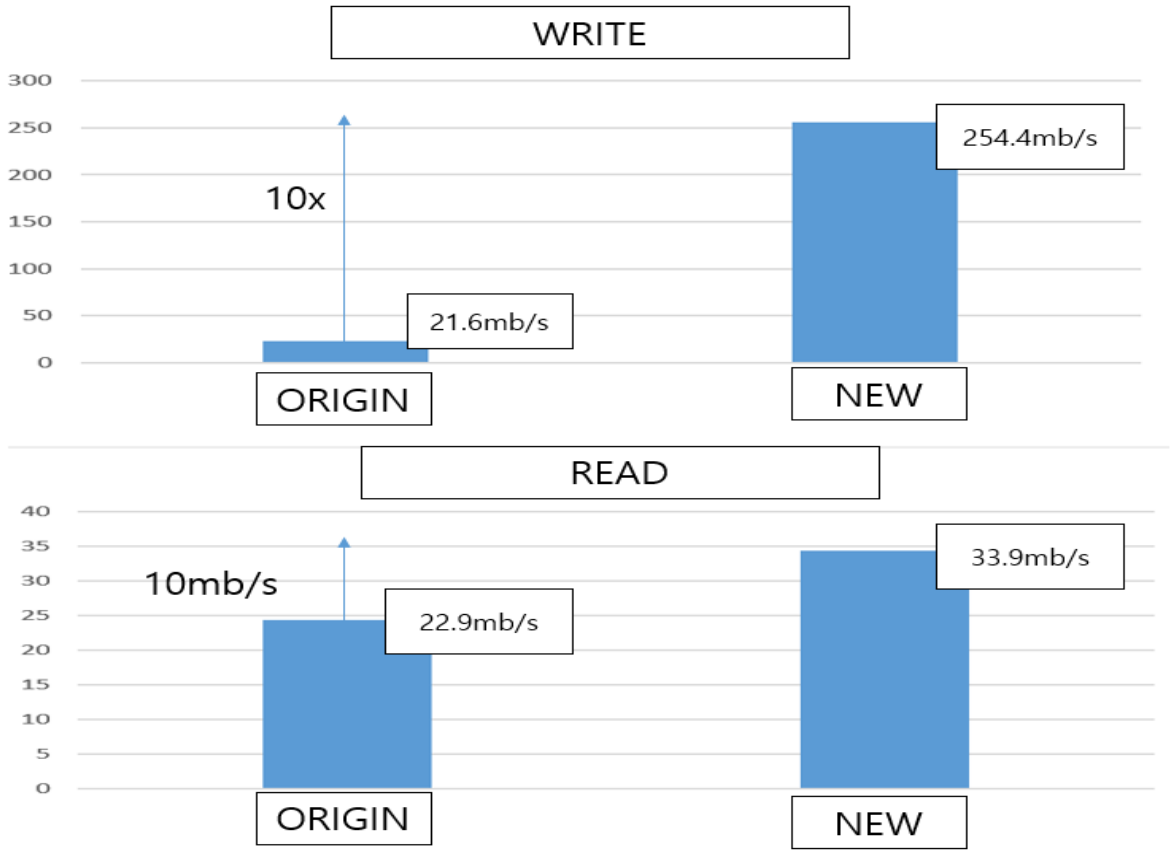
따라서 version1을 기준으로 제시 되었던 시스템에 맞는 프로그램을 만들도록 하였다.



5-2 ORIGIN NEW 성능 비교



VERSION 1으로 만든 시스템을 올렸을 때 기존의 시스템과 비교를 하는 부분의 캡처이다.



WRITE 연산에서의 성능이 두드러졌던 이유는 같은 LSM TREE 구조를 사용하지만 기존의 ROCKS DB에서 사용하는 LSM TREE에서는 COMPACTION연산이 일어날 때 데이터까지 읽어 들여 COMPACTION 과정을 진행하여 한번 할 때 참여하는 LEVEL의 노드 개수 * SKIPLIST SIZE 만큼의 데이터가 메모리에 올라가야 되지만 새로 만들어진 LSM TREE에서는 SKIPLIST SIZE 만큼이 아닌 HEADER(META DATA) SIZE를 가지고 연산을 해 빨라 진 것 같다. 또한 PAGE FTL을 사용하는 기존의 시스템의 H/W spec이 RAM 1GB이고 이는 모든 ENTRY 테이블을 올릴 수 없고 스위칭이 일어난 다는 것을 의미한다. 즉 FTL에서의 이런 추가 적인 연산도 영향이 있었을 것으로 예상된다.

5-3 memory 비교

PAGE FTL에서의 메모리는 1TB 기준 1GB 정도 차지하게 된다. 우리의 목표는 이 보다 적은 메모리를 사용하는 것이 목표였다.

VERSION 1에서는 앞서 설명했듯이 Bloom filter를 사용하지 않기 때문에 메모리가 훨씬 줄어들게 되는데 1TB일 때 8MB의 Skiplist Size로 저장이 된다면 약 25만개(2^17)의 B+ TREE의 ENTRY가 생기게 된다. 이때 완성된 ENTRY 하나의 크기는 다음과 같다.

```
struct Entry; struct Node;
typedef union Child{
    struct Entry *entry;
    struct Node *node;
}Child;

typedef struct Entry{
    KEYT key;
    KEYT version;
    KEYT end;
    KEYT pbn;
    struct Node *parent;
}Entry;

typedef struct Node{
    bool leaf;
    short count;
    KEYT separator[MAXC];
    Child children[MAXC+1];
    struct Node *parent;
}Node;
```

SIZEOF(KEYT) * 4 + SIZEOF(POINTER) =36byte (:KEYT가 8byte일때)

즉 2^17*36byte임으로 8MB의 메모리를 차지하게 된다.

ENTRY 관리를 위한 NODE 개수는 다음과 같이 구할 수 있다. B+트리 정의에 의해 각 NODE는 최소 반 이상의 자식이 차있어야 됨으로 자식의 최대개수 MAXC를 반의 개수만큼의 자식이 들어있다고 칠 때 이때의 NODE 개수는 MAXC가 8일 때 (2^17)/(MAXC/2)=2^15개의 개수만큼 들어간다. Leaf Node의 개수는 2^15일 때 모든 NODE 들의 개수는 Leaf Node의 개수의 2배 보다 적음으로 최대 2^16개의 Node가 생성된다.

NODE SIZE : 1byte+2byte+8*8byte+8*9byte+4byte=143byte

총 NODE SIZE = (2^16)*143byte는 대략 10*2^20byte가 나오며 10mbyte의 크기를 가진다.

즉 VERSION1에서 필요한 memory size는 30mb 이하이며 이는 1GB의 메모리를 필요로 하던 기존의 PAGE FTL에 비해 월등히 좋아진 결과이다.

VERSION2 에서는 위에서 계산했던 VERSION1의 메모리 양에서 bloomfilter의 memory 양을 더해주면 된다. 총 ENTRY 개수 만큼의 bloomfilter가 들어 있고 bloomfilter의 사이즈는 skiplist의 key 개수와 k값의 곱으로 나타낼 수 있음으로 1024*6bit이다 이 값을 8로 나누어 byte로 변환하면 800byte가 된다. 즉 bloomfilter로 인한 메모리는 (2^17)*(800)byte 이고 이는 2^27byte정도가 됨으로 약 128mb를 차지하게 된다. 즉 180mb 정도의 메모리를 차지하게 되며 기존의 1G보다

1/5로 줄어든 것을 확인 할 수 있다.

완성 동영상 URL: https://www.youtube.com/watch?v=GEV13ocdUtY

완성 프로그램 git : https://github.com/kukania/lsmtree

6. 결론

6-1 향후 활용 방안

현재 개발된 프로그램은 NOHOST 라는 논문의 주제 시스템의 핵심 프로그램으로 들어갈 것이다. 이러한 시스템을 담은 서버들을 여러 개 연결하여 분산시스템에서 동작의 원활함을 판단한 후 논문을 작성할 예정이다.

또한 만들어진 LSM-TREE FTL 은 원래의 목적이었던 SSD 가상화 시스템임 BlueDBM 에 올릴 수 있도록 다듬고 기존의 FTL 과의 성능 비교를 통해 새로운 FTL 로서의 가치가 존재하도록 만들겠다.

6-2 프로젝트 후기

임준수

3~4 개의 프로젝트를 진행했지만 모든 프로젝트들이 High Level 인 즉 client 측과 닿아 있는 부분의 프로젝트만을 진행해서 그런지 Low Level 인 Device 나 Device 와의 연관관계를 고려해야 되는 프로젝트가 어색하기만 했습니다. 처음 제안서를 쓸 때만 하여도 쉬울 것이라 여겨 충분히 기간 내에 끝낼 수 있으리라 생각했습니다. 하지만 중간에 큰 어려움이 닥쳐 방향을 틀어야 했고 그러한 점이 너무나도 아쉽게 다가왔습니다.

지금 생각해 보면 개발 이외의 부분을 너무나 소홀히 여긴 것 같습니다. 제안서나 발표자료, 또 발표. 개발이 주된 역할이라 생각했기에 개발 외적인 것에 소홀히 하였고 그로 인해 학우 분들 및 제 발표를 들어주시는 사람들에게 너무나도 정보전달을 못해주어 아쉬웠습니다.

여러 아쉬움을 남긴 프로젝트였지만 결국에는 목표하던 바에 가까스로 도달했기에 좋은 기회였다 생각합니다.

김근표

처음으로 제안서나 보고서를 작성해보면서, 문서화 작업의 중요성을 느끼게 되었습니다. 여러가지 팀 프로젝트를 진행해보았지만 이전에는 구체적인 문서화 작업을 하지 않았고, 또한 필요성도 느끼지 못했습니다. 하지만 졸업프로젝트를 진행하면서 문서화 작업을 통해서 팀원들

간의 소통도 더 원할해지고, 정량적 목표를 설정함으로써 프로젝트 진행에도 도움이 되었던 것 같습니다.

또한, 발표의 중요성도 느꼈던 것 같습니다. 특히, 마지막 수업시간에 많은 팀들의 발표를 들으면서, 어떤 조의 발표는 명확하지만 어떤 조의 발표는 그 팀이 개발했던 것을 잘 어필하지 못했던 것 같습니다. 아는 것과 설명하는 것은 다른 것이라고들 하는데, 아는 것 뿐만 아니라 설명하는 것의 중요성을 깨닫게 되는 계기가 되었습니다.

문한별

실제 프로젝트를 진행하니 설계시에는 예상하지 못한 부분이 많이 드러났습니다. 발표를 들으니 저희만 그랬던 것이 아니고 설계과목을 수강한 모든 조가 그랬습니다. 이것은 아직은 미숙하기 때문에 생기는, 지극히 당연한 일들이라고 생각합니다. 사회에 나가서 겪게 될 일들을 작게나마 경험할 수 있다는 것에 용기를 얻을 수 있었습니다. 또한 3~4 개월이라는 비교적 짧은 기간과 그 기간에 맞추어 프로젝트를 계획하려다 보니 좀 더 정교하게 개발을 할 수 없었던 점이 아쉬웠습니다. 그래도 좋은 조원들과 함께 할 수 있어서 행복했습니다.

6-3 다음 학기 수강생 들에게

임준수

가장 중요한 것은 내가, 또 우리 팀원들이 무엇을 얼마나 개발해 봤는지에 대한 개발 역량이 제대로 평가되어야 한다는 점을 알았으면 좋겠습니다. 무모하게 제안서를 작성하고 큰 벽에 막혀 열정이 식는 것이 가증 큰 벽인 것 같습니다. 또한 발표나 제안서 등과 같은 개발 외적인 부분도 중요히 여겼으면 합니다. 아무리 좋고 잘난 프로그램을 만든다 하여도 그것을 알아 주지 못하면 자기만족일 뿐입니다. 그것을 잘 전달 하는 것도 중요한 부분인 것 같습니다.

김근표

음 학기 수강생들은 조금 더 다양한 분야에 도전하였으면 좋겠습니다. 대부분의 학생들이 응용 어플리케이션 개발이 대부분이었습니다. 따라서, 사용하는 기술들이나 툴들도 거의 비슷했던 것 같습니다. 다양한 분야에서의 도전이 다른 학생들에게도 자극이 되고, 또 이러한 자극에서 새로운 아이디어가 나올 수 있을 것이라고 생각합니다.

문한별

설계 시 최대한 많은 점을 고려했으면 좋겠습니다. 또한 모바일 어플리케이션 부분을 대다수의 조가 진행했는데, 생각해보면 모르는 많은 분야가 있을 것이라 생각됩니다. 분야에 대해 알아가면서 설계를 진행한다고 생각하고, 도전적인 자세로 임했으면 합니다.

7. 참고 문헌

<논문 참조>

(a) *Pugh, William (April 1989). Concurrent Maintenance of Skip Lists (Technical report). Dept. of Computer Science, U. Maryland. CS-TR-2222.*

<그림 및 이론 참조>

2 장 - https://en.wikipedia.org/wiki/Write_amplification (2017.06.11)

(1) http://tech.kakao.com/2016/07/15/coding-for-ssd-part-3/ (2017.06.11)

(2) https://en.wikipedia.org/wiki/Write_amplification (2017.06.11)

3 장

(1) http://www.redisgate.com/redis/configuration/internal_skiplist.php(2017.06.11)

(2) https://ko.wikipedia.org/wiki/B%2B_%ED%8A%B8%EB%A6%AC(2017.06.11)

(3) https://ko.wikipedia.org/wiki/%EB%B8%94%EB%A3%B8_%ED%95%84%ED%84%B0(2017.06.11)

4 장

(1),(2),(3) http://d2.naver.com/helloworld/749531 (2017.06.11)

(4) http://www.redisgate.com/redis/configuration/internal_skiplist.php (2017.06.13)

(b) https://shrikantbang.wordpress.com/2014/04/22/size-tiered-compaction-strategy-in-apache-cassandra/ (2017.06.13)

(c) http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra (2017.06.13)

(d) https://github.com/chamdoo/bdbm_drv (2017.06.14)

(e) ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide

(https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf)