종합설계 제안서

1. 주제 정의

- 최종 목표

SSD에 사용되는 기존의 FTL을 B+ Tree와 Skiplist로 이루어진 LSM Tree로 구성하여 SSD의 성능 향상.

2. 배경 (아래 아이템들의 구성 순서는 변경 가능)

- 동기: 개발 이유

기존의 SSD의 FTL의 단점.

기존의 SSD에서 사용되는 FTL의 경우에는 여러 가지 단점을 지니고 있다. Mapping Table을 지니고 있어야 한다는 점, delete가 Block단위로 동작하기 때문에 Garbage-collection이 필요하다는 점, 또한, NAND Flash의 특성상 Wear Leveling을 필요로 한다는 점이다. SSD의 FTL을 개발하고자 하는 LSM Tree로 구성할 경우에 위에 제시된 단점들을 보완할 수 있다.

우선, 기존의 FTL은 크게 두 가지의 기능을 지니고 있다. 하나는 논리주소와 물리주소의 Mapping이고 다른 하나는 Garbage-collection이다.

논리적인 블록 맵핑은 호스트 영역의 논리 주소(LPA, Logical Page Address)를 NAND 플래시 메모리의 물리적 주소(PPA, Physical PageAddress)로 변환해주는 역할을 담당한다. 이러한 역할을 수행하기 위해서는 LBA를 PBA로 변환해주는 Table을 주로 사용한다. 이 Table은 meta data로써 실질적인 DISK로 사용 가능한 영역이 아니다.

SSD에서의 쓰기와 읽기는 Page단위이지만 삭제는 Block단위로 이루어진다. 이러한 특성 때문에 Page overwrite연산을 수행하지 못한다. 페이지 단위의 수정을 위해서는 기존 page를 Invalid상태로 두고 다른 영역에 수정된 page를 쓰고 Mapping Table의 PPA를 바꾸어줌으로써 수행 가능하다. 하지만 이러한 방식은 Invalid한 데이터가 쌓이게 되면 저장공간이 줄어들어 결국 가득 차게 된다. 이 때, Garbage-collection을 통해서 Invalid한 데이터들을 정리하여 줄여주어야 한다. 하지만 이러한 연산은 데이터를 모으고 지우고 써야 되기 때문에 상당히 오래 걸리는 연산이다.

NAND Flash의 특성 때문에 SSD의 DISK는 Block마다 접근 횟수의 제한이 존재한다. 위에서 언급한 Garbage-collection은 여러 번의 DISK접근으로 이루어진다. 따라서 이 과정에서 다른 Block 보다 먼저 이 횟수를 소진하는 경우가 발생할 수 있다. 그렇게 되면 해당 용량만큼 DISK의 크기가 줄어드는 현상이 발생하게 된다. 이를 막기 위해서 SSD에서는 Wear Leveling을 위해서 이러한 Block접근 횟수를 고려해주어야 한다.

- 이 프로젝트 완료로부터 기대되는 효과 (예) 사회적 기여, 시장 상황 등 설명

FTL을 LSM Tree로 구성함으로써 얻게 되는 효과

FTL을 LSM Tree로 구성하게 되면 위에서 제시된 단점들을 보완해줄 수 있다.

우선, LSM Tree로 구성하게 되면 큰 크기를 차지하는 Mapping Table이 사용되지 않기 때문에 실제 사용 가능한 DISK 사이즈를 더 확보할 수 있다.

두 번째로, LSM Tree로 구성하게 되면 Page단위의 쓰기가 아닌 Block단위의 수정이 한번에 이루어지기 때문에 Garbage-collection이 필요하지 않게 된다. 그 덕분에 순서대로 DISK를 사용하면 되기 때문에, Wear Leveling 또한 자동으로 되게 된다.

- 유사 프로젝트 검색 및 해당 프로젝트 결과물들의 문제점

예상되는 단점.

하지만 SSD의 FTL을 LSM Tree로 구성하게 되었을 때의 단점도 존재한다. 크게 읽기 속도 저하와 DISK의 중복된 데이터 존재이다.

기존의 SSD의 FTL의 경우에는 Mapping Table을 이용하기 때문에 자료에 대한 읽기 속도가 빨랐다. 하지만 LSM Tree로 구성하게 되면 별도의 논리적 블록 맵핑 과정이 필요하다. LSM Tree내에 존재하는 Page는 굉장히 빠르게 접근 가능하지만 그렇지 않은 경우에는 많은 Block을 탐색하여 Mapping 정보를 찾아야 하기 때문에 읽기 속도가 기존보다 느리다.

또한, B+ Tree의 경우에 여러 개의 Step으로 구성하는 것이 효율적이다. 이 때문에 같은 Mapping 정보가 여러 Step에 걸쳐서 존재할 수 있다. 이러한 데이터는 중복을 일으키고 명백한 데이터 낭비이다.

3. 개발 내용

- 개발 플랫폼: bluedbm(SSD 가상 디바이스), Linux kernel 4.x, GCC 컴파일러
- 상세 요구사항

. 기능적 요구사항 (functional requirements)

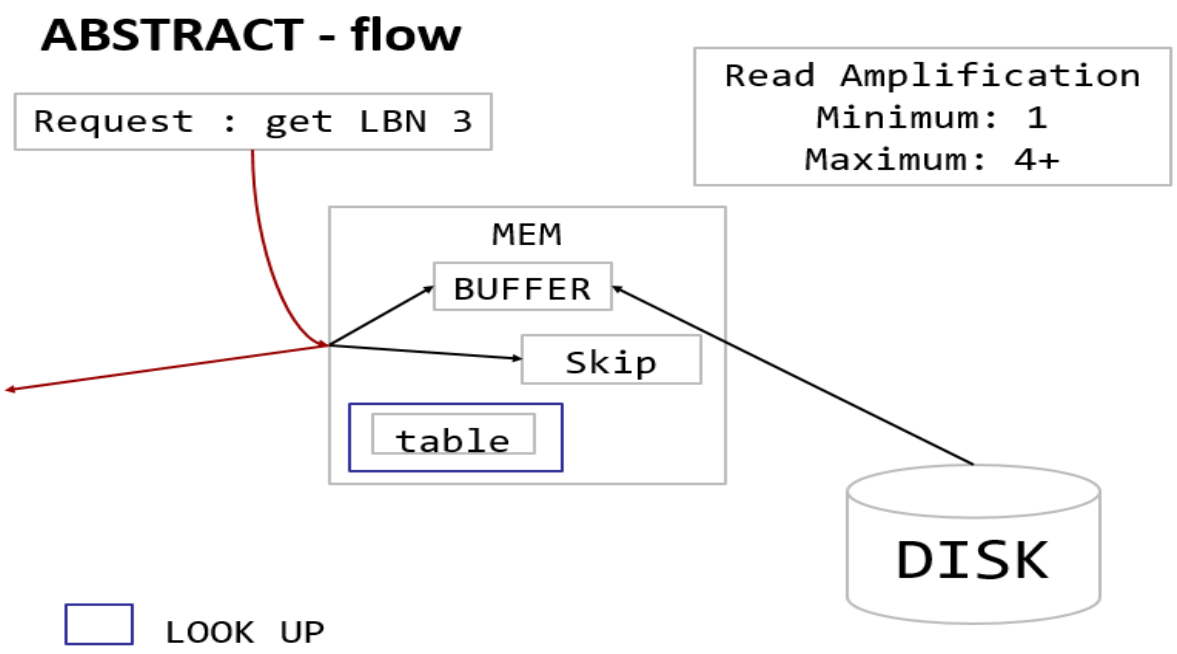
Linux file systme에 맞는 read, write, create, delete 연산을 제공.

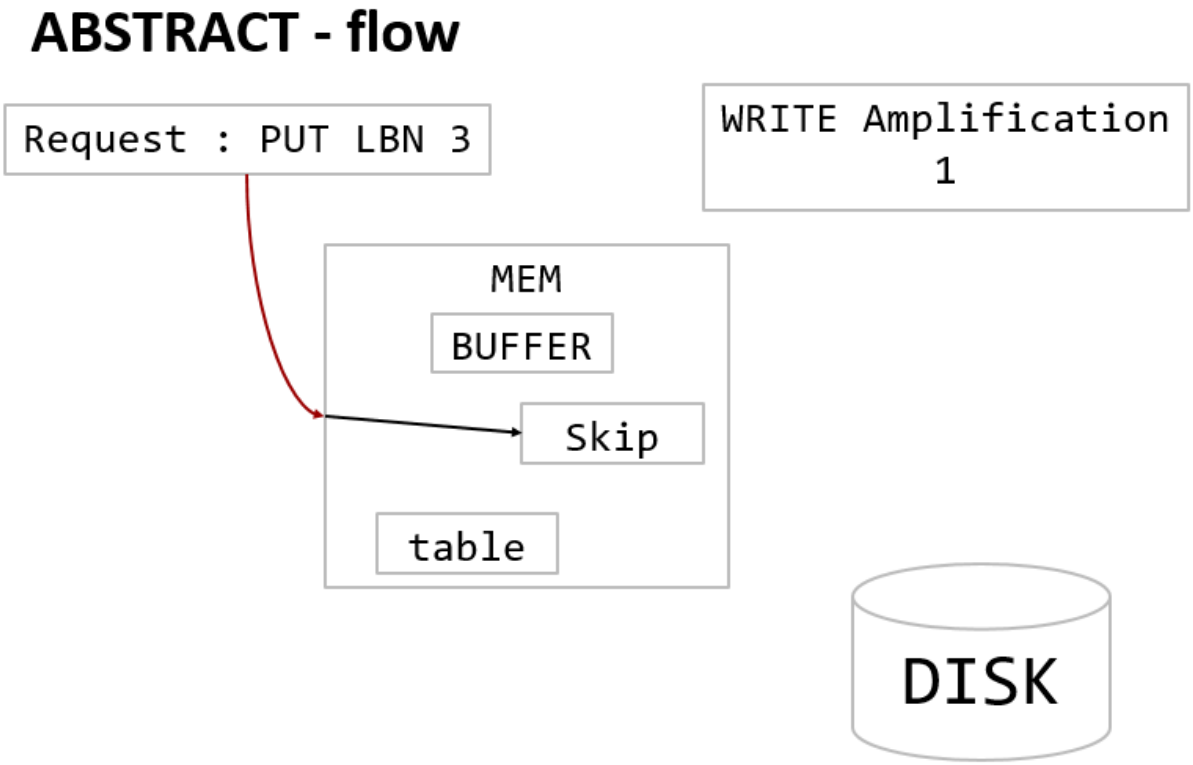
. 비기능적 요구사항 (non-functional requirements)

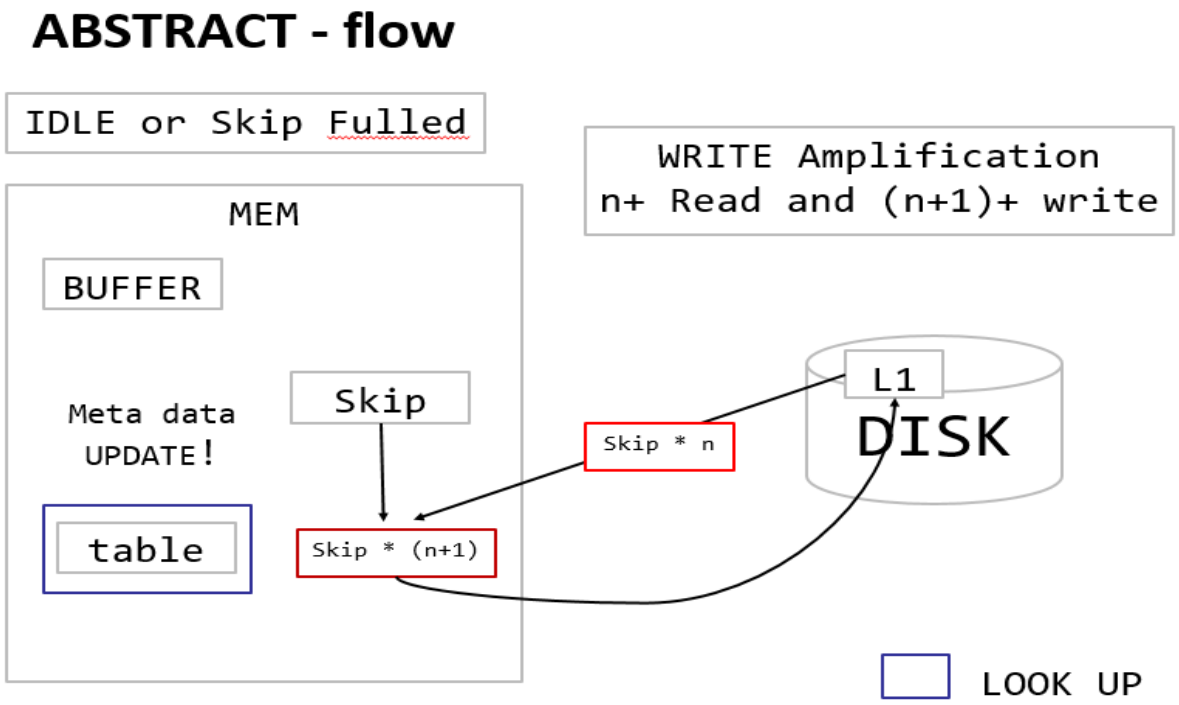
기존의 Flash Translate Layer보다 높은 반응 속도를 가지는 write 연산과 기존의 방식과 큰 차이가 없는 read 연산의 반응속도. 기존의 방식보다 조금 더 많은 양의 Storage 공간.

- 상세 구조 (system components and relation between components)

명령어 흐름

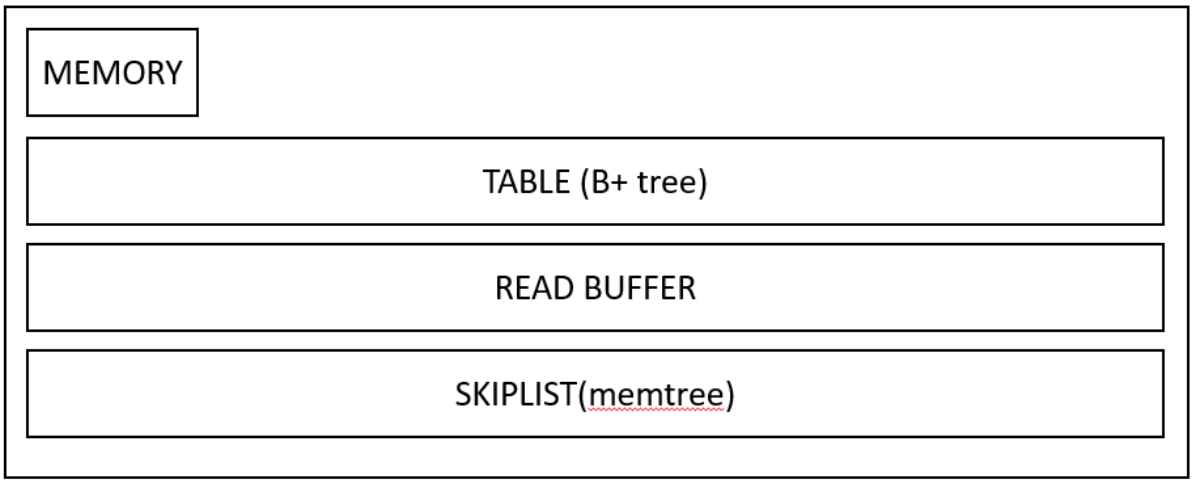


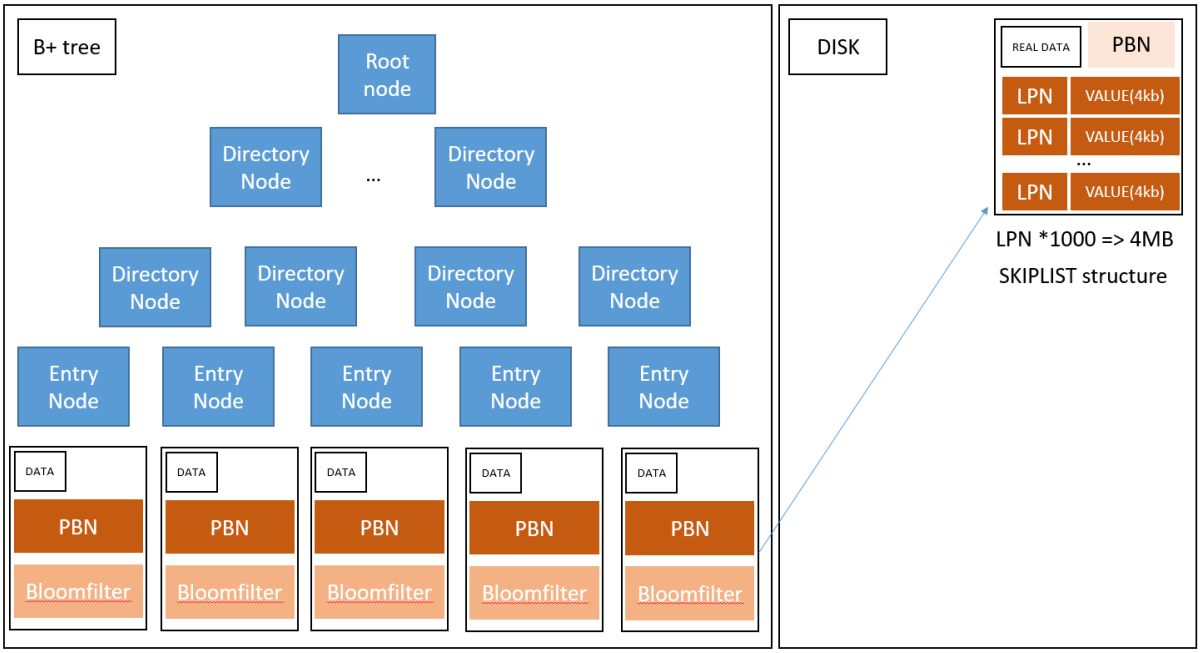




4. 개발 방법

LSM tree를 이용한 FTL구현에서는 B+ tree와 Skiplist 그리고 bloomfilter 크게 세가지 컴포넌트들이 사용된다. 하지만 FTL의 특성상 delete 연산이 존재하지 않기 때문에 기존의 B+ tree와 Skiplist의 구조가 달라질 수 있다. 또한 SSD 내부의 메모리 위에서 동작을 하기 때문에 프로그램의 사이즈가 중요한 요소가 될 수 있다. 이러한 특성 때문에 세가지 컴포넌트 전부 직접 구현하기로 하였다. 개략적인 LSM트리의 구조는 다음과 같다.





LSM tree를 구현한다면 write연산은 가장 처음에 메모리에만 있는 Skiplist에 쓰여지게 된다. 이때 만약 skiplist가 가득차게 된다면 그제서야 B+tree로 내려가게 된다. B+tree의 구조의 데이터들은 위의 그림과 같으며 실제 파일의 데이터는 Disk에 저장되게 된다. 실제 데이터가 저장되는 단위는 4mb로 원래 disk에 저장되는 단위인 4kb와 비교를 하게 된다면 1000개의 LPN을 가지게 된다. 이는 Skiplist의 크기가 되기도 한다.

B+ tree의 data node는 실제로 저장된 데이터의 위치를 가르키게 되는데(PBN - value) 이때 data node의 key 값은 PBN에 적혀진 1000개의 LPN(Logical page number)의 최소 값이 된다. 또한 B+ tree는 단일로 존재하지 않고 여러 개로 존재하는 각각의 B+ tree를 Level로 칭하게 된다. 같은 맥락으로 Skiplist는 Level 0로 불리고 Disk에서는 Disk의 용량에 맞게 Level 1, Level 2...식으로 늘어져 간다.

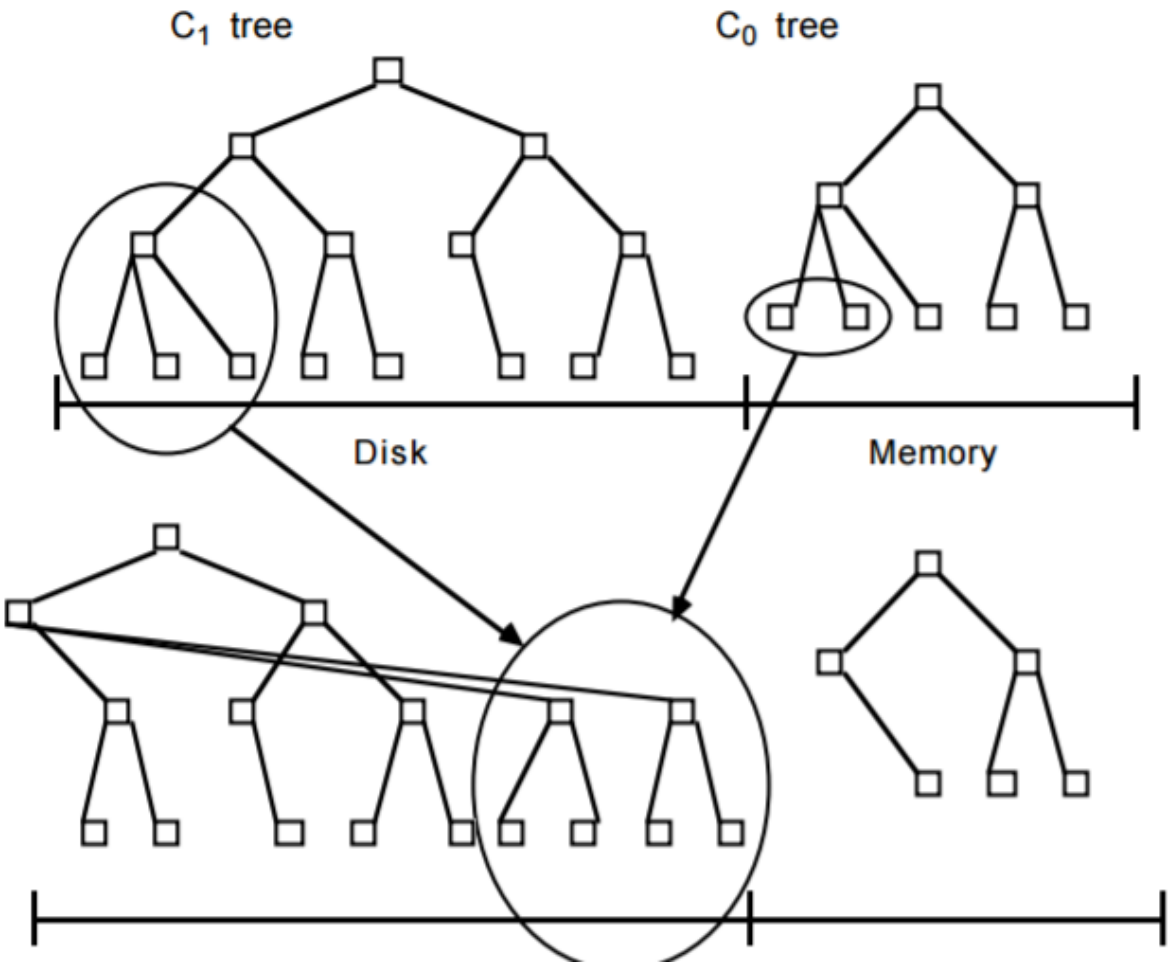
앞서의 설명은 Level 0(Skiplist)가 가득차면 Level1(disk)로 내려주게 되는데 이는 Level 1에서와 Level 2(disk에 존재)에서도 마찬가지이다. 단지 Level 별로 다른 점은 각각의 크기가 다르다는 것인데 Level 0 에서 Level N으로 갈수록 같은 배수(R)만큼 점점 커진다. 이 때 Level의 개수인 N과 R은 Merge 연산(Compaction 연산이라고도 불림)에 trade off 관계가 생기게 된다. N이 커질수록 R은 감소하고 Merge 연산의 호출은 줄어들지만 한 번의 Merge연산은 커지게 된다. 이는 Write가 작게 여러 번 오는 시스템에서는 유리한 반면 반대의 경우에는 효율이 좋지 않게 된다. N이 작아지면 R이 커지게 되고 이는 반대의 경우를 초래하게 된다.

(R은 각 level 마다 동일해야 함은 LSM tree 논문에 자세하게 나와 있다.- The Log-Structured Merge-Tree (LSM-Tree) Patrick O'Neil)

모든 것은 동적으로 설정할 수 있게 하는 것이 목표로 Skiplist의 사이즈와 N과 R 등등을 사용자 설정에 맞출 수 있도록 하는 것이 한 가지 목표이다.

Merge 연산은 크게 보면 두 가지 경우가 있을 수 있다. 우선 Merge 연산이 호출되는 과정을 알아보면 어떤 level이 가득 차게 되면 merge 연산이 실행되고 그 실행의 결과가 다음 level로 내려가는 결과로 이어지게 된다.

<merge 연산을 보여주는 그림 – 결과로 새로운 곳에 데이터가 적히게 된다>



첫 번째 방법은 데이터가 더 낮은 level로 내려가게 될 때 해당 데이터가 가진 범위(block당 1000개의 LPN)를 찾아 갱신을 해주는 방법이다. 모든 level은 key값으로 정렬이 되어 있음으로 merge sort와 비슷하게 연산을 진행하면 될 것이다. 이때 두 level에서 같은 key값을 가지는 entity는 높은 level의 데이터로 덮어 쓰게 된다. 결과적으로 높은 level의 데이터 하나(4MB)를 내릴 때 최악의 경우 모든 하위 level의 데이터를 읽어야 하고 그 만큼의 데이터와 내려올 데이터를 다시 써주어야 하는 경우가 발생한다.

위의 경우에는 merge 연산을 진행해야 될 경우 연산의 수행 속도는 최악으로 따졌을 때, 낮은 level의 size가 S라 했을 때 disk 읽기 연산이 S번 필요하고 merge 연산이 S*1000+1000이 disk write연산이 S+1번 발생한다. 이 결과로 read연산을 진행할 때 각 level안에서는 중복된 key가 존재하지 않음으로 read연산의 효율이 좋아질 것으로 예상된다.

반면, 다른 방법은 낮은 level로 보낼 때 <level 안에서 중복된 key가 존재할 수 있다>라는 전재

로 진행된다. level이 가득 차게 되면 같은 level의 데이터 정해진 개수를 묶어서 낮은 level로 보내지게 되는데 이때의 merge 연산은 그 데이터들 사이의 중복된 key값을 재거 하고 낮은 level로 보내 지는 것이다.

이 때 merge 연산에 사용되는 연산은 한 번에 merge를 하게 될 데이터 단위를 K라 하였을 때 K번의 읽기와 K*1000개의 entry에 대한 merge sort 작업 그리고 K개의 disk 쓰기 연산이 필요할 것이다. 앞서 말한 방법의 S와 K는 단위 자체가 10배 이상 정도 차이가 나기 때문에 이 방법을 사용했을 때는 merge연산이 빠르게 진행 될 수 있다는 장점이 있다. Merge 연산이 빠르다는 것은 LSM tree로 들어오는 쓰기 Request에 대한 처리를 빠르게 할 수 있다는 의미와 같다.

단 read연산을 할 때 같은 level에도 같은 key 값이 존재할 수 있기 때문에 최신의 데이터를 찾기 위해서는 B+ tree의 데이터 들의 key값에 시간이나 version과 같은 최신의 데이터라는 것을 알 수 있는 추가적인 정보를 필요로 하게 된다.

즉 두 가지 방법은 read와 write에 대해서의 trade off관계가 있으며 어떠한 시스템에서 무엇이 잘 돌아가는지에 대한 것은 구현 결과의 bench marking을 통해 알아볼 필요가 있다. 우선 적인 목표는 SSD H/W특성상 2번째 merge를 구현하는 것이 조금 더 이득일 것이라 생각되어 2번째 방법을 구현하는 것을 목표로 두고 있다.

세가지 컴포넌트가 구현이 끝나면 추가적으로 LSM tree를 만들기 위한 컴포턴트 별의 특수한 연산을 구현해야 된다.

Skiplist의 경우 직접적으로 쓰여지는 단위 이기 때문에 SSD의 channel, way, block, page라는 구조에 맞추어서 최대한 병렬성을 활용하기 위한 file header나 구조가 필요하고 읽기 역시 그 file header를 읽고 정확한 데이터를 가져오는 연산이 필요해 진다. (skiplist_read, skiplist_write)

B+ tree에서는 merge연산에서 하나의 데이터가 아니라 여러 개의 범위 안의 데이터를 읽기 때문에 range_find라는 새로운 연산이 필요할 것이다. 또한 이것을 최신 순으로 정렬을 하여 결과를 주는 방법도 필요하기 때문에 시간 값을 key 값에 어떻게 표현하고 사용할지에 대한 설계가 필요하다.

Bloomfilter는 LSM tree를 위한 새로운 연산이 필요하지 않다.

위의 모든 컴포넌트 및 LSM tree의 작업이 끝난다면 간단한 테스트를 통해 LSM tree가 완성이 되었는지에 대해 알아볼 것이다. 우선 하나의 파일을 열어 그 파일을 SSD라 생각하고 가상으로 공간을 나누어 put(write),get(read) 연산을 수행해 sequence, random에 관한 read,write연산이 제대로 작동하는지 확인을 할 수 있을 것이다.

테스트가 끝나면 FTL로서의 테스트는 SSD가 가상화 되어있는 github의 Bluedbm 프로젝트를 사용할 것이다. (https://github.com/chamdoo/bdbm_drv) 이 프로젝트는 메모리를 Bluedbm이라는 SSD구조를 가상화한 것으로 자신의 FTL을 올려서 테스트 할 수 있다. 그렇게 하기 위해서는

interface를 맞추어 주어야 하는데 가장 많이 쓰이는 Bluedbm 안의 Page Ftl을 참고하면서 맞추어 갈 수 있을 것이다. 이 때 POSIX로 구현된 LSM tree는 kernel에서 돌아가지 않는 함수들(malloc,free etc...)등등을 kernel에 맞는 함수로 바꾸어서 인터페이스를 맞추어야 하기 때문에 시간이 꽤 걸릴 것으로 예상된다.

기존의 FTL에 비해서 메모리에 상당한 양의 데이터가 disk에 쓰여지지 않고 저장되기 때문에 power failure에 대해 취약할 수 있다. 실제적으로 SSD 위에서 돌아가는 FTL들은 이러한 문제를 약간이나마 전부 가지고 있기 때문에 그에 대한 해결책은 그에서 차용해 다음과 같이 보안을 할 수 있을 것이다.

특정 Disk 영역을 Log 영역(최소 8mb)으로 잡아 write request에 대한 것들을 request를 받자 마자 disk에 적어주는 것이다. 8mb로 하는 이유는 첫째로 메모리에 올라가는 데이터의 양이 4mb라는 점이고 다른 것은 ssd의 지우기 단위가 4mb라는 것이다. 즉 8mb가 다 찰 때 쯤이면 4mb block 중에서 처음으로 쓰여져 있는 것이 모두 invalid 한 것이기 때문에 한번에 지워 버릴 수 있어 효율적 측면에서 보았을 때 8mb가 적당하다고 판단 되었기 때문이다.(4mb는 SSD의 block단위로 치환 될 수 있다.)

이러한 power failure에 대한 이슈는 주제에 포함되는 것이기는 하지만 최우선 목표가 아니기 때문에 우선적인 목표를 끝낸 뒤 논의해 볼 수 있을 것이다.

5. 개발 계획

예상 소프트웨어 크기 : 10mb

예상 생산성 : 200 LOC/MM(man-month)

예상 비용 : -원

각 과정 간 컴포넌트 개발은 서로 독립적이므로 팀 구성원이 병렬로 프로그래밍을 할 수 있다. 다만 각 컴포넌트 간 인터페이스를 맞추는 데 시간이 많이 소요 될 것으로 보인다. 따라서 컴포넌트 간의 인터페이스에 중점을 두기로 한다.

예상 생산성의 경우, C language 로 구현을 하고 각 컴포넌트 별 프로그래밍 난이도가 차이가 나기 때문에 평균으로 계산한다.

예상 비용의 경우, 매우 적거나 없다. 개인의 도구를 이용하여 작업하고 S/W tools의 경우 무료로 이용이 가능하다. 또한 성능 측정 방식을 SSD H/W에 이식하는 것이 아닌 시뮬레이터로 하므로 SSD구매 비용이 발생하지 않는다.

주요 개발 계획은

1. LSM Tree 구현을 위한 주요 컴포넌트인 bloom filter, skip list, B+ tree 를 병렬적으로 구현
2. 각 컴포넌트의 인터페이스를 맞춤
3. 시뮬레이터에 프로그램을 올린 후 성능을 측정

4.1 예상결과와 실제 결과에 대한 비교.분석 및 추가적 성능향상을 위한 프로그램 수정

4.2 예상하지 않은 결과가 나왔다면 원인 분석 및 프로그램 수정

Ps. 각 과정 간 회의 및 문서화를 통한 생산성 저하 방지

주요 중간 목표들 :

- Bloomfilter 구현
- Skiplist 구현
- B+ tree 구현
- LSM tree 구현
- 가상 디바이스와의 인터페이스 조정
- 기존 소프트웨어와의 비교
- 최종 결과물 평가를 위한 평가 기준

정량적 목표 :

1TB의 SSD 저장공간에서 1%이하의 metadata 용량

read 연산 시 기존 SSD에 비해 속도의 향상

write 연산 시 기존 SSD에 비해 속도의 향상

6. 참고문헌

The Log-Structured Merge-Tree (LSM-Tree) Patrick O'Neil (http://www.cs.umb.edu/~poneil/lsmtree.pdf)

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