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 두 가지의 기능을 지니고 있다. 하나는 Logical Block Mapping이고 다른 하나는 Garbage-collection이다.

논리적인 블록 맵핑은 호스트 영역의 논리 주소(LBA, Logical Block Address)를 NAND 플래시 메모리의 물리적 주소(PBA, Physical Block Address)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 LBA를 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.

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

- 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.