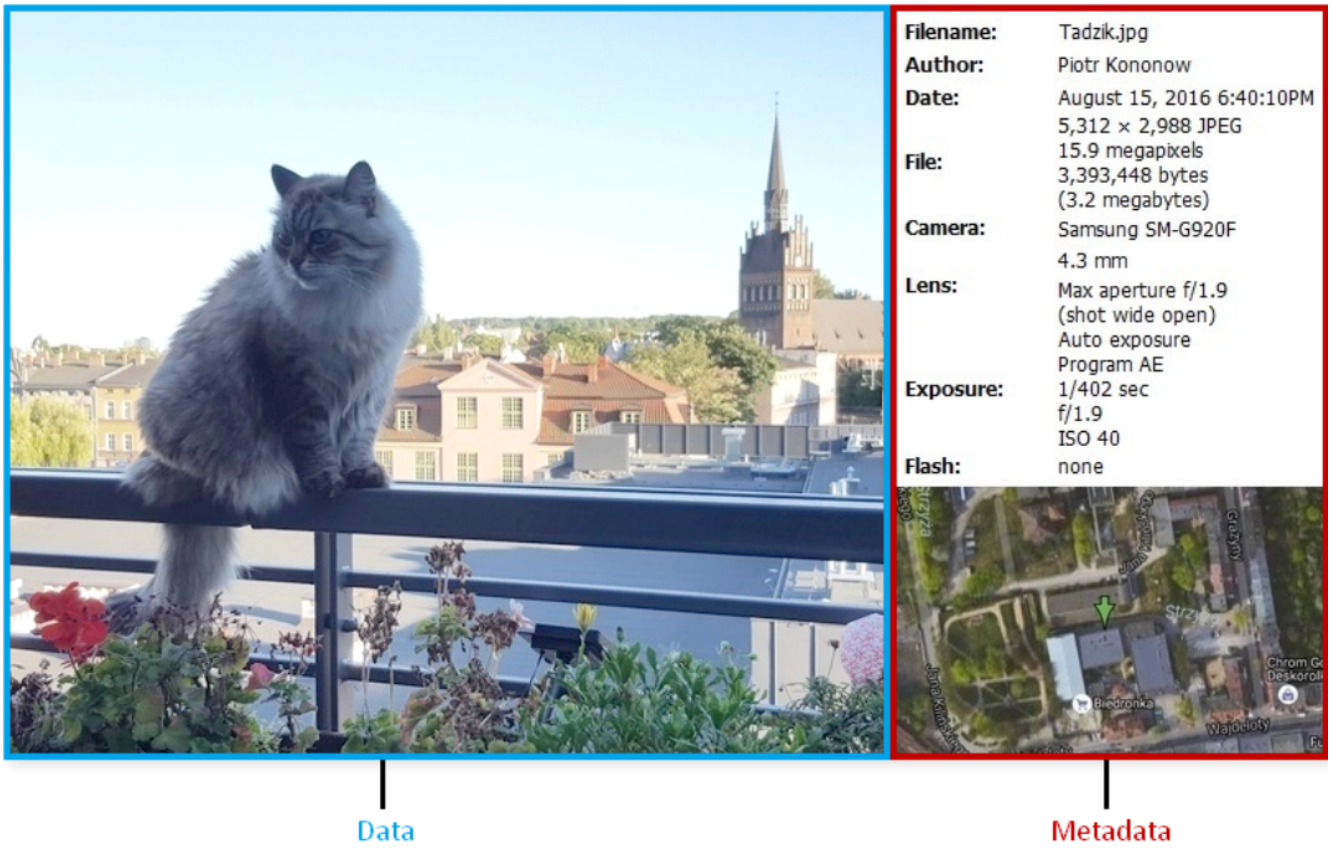
# 1. INTRODUCTION

- CPU의 처리 속도가 빠르게 증가하고 있지만 storage는 CPU의 발전과 대비하면 발전이 많이 느리는 현황입니다.
- log-sturctured file system은 이런 가설을 했습니다: file이 cache memory에 쓰이고 cache가 많아지면 read성능이 좋아집니다. 결론으로 storage의 기존 문제점이 random write인데 log방식으로 sequential write로 바꾸게 되며 write 효율이 좋아지면서 unix file system보다 더 빠른 회복속도를 보여줍니다.

# 2. DESIGN

1. Technology
    - CPU
        - CPU의 성능은 매년 거의 2배의 속도로 증가하고 있습니다.
    - Storage
        - Storage의 기술도 증가하고 있지만 주로 cost과 용량 방면으로 돼 있습니다. 하지만 성능은 bandwidth와 seek time으로 결정되는데 이 부분은 크게 발전하지 않았습니다. (bandwidth는 raid등 방식으로 올릴수 있지만 seek time은 disk의 구조상 한계가 있습니다.)
    - Memory
        - Memory의 용량도 CPU와 비슷한 속도로 2배씩 증가하고 있습니다. 현대 file system은 memory에 hot data를 저장하고 있고 더 큰 memory를 사용하면 더 많은 hot data를 저장할 수 있습니다. 하지만 이러면 cache에서 read조작이 많아지고 disk에서는 write조작의 비율이 높아집니다. 이러면 disk의 성능의 bottle neck는 write가 됩니다.
        - 더 큰 memory를 사용하면 write buffer로 사용할 수 있고 write가 더 효율적으로 진행 되지만 이 와 같이 충돌이 발생하면 buffer에 있던 data가 손실될 수 있습니다.(해결책의 하나로 NON-VOLATILE RAM을 사용할 수 있습니다.)
2. Workloads
    - Metadata




이런 data의 속성,block의 주소등 information을 metadata라고 합니다.

- Office 혹은 engineering 환경에서는 file의 크기가 작고 file의 수가 많아 대량의 random I/O가 발생합니다. 이런 환경에서 data를 삭제혹은 생성하면 time cost는 metadata의 update에 의해 결정됩니다.
- 이와 반대로 큰 file을 사용하는 환경에서는 file allocation policies에 의해 metadata update의 영향을 받지 않습니다.
- 그래서 Sprite LFS는 작은 파일을 효율적으로 가져오는 데 주력합니다.

3. Two Problems with Existing File Systems
   - Unix FFS에서는 information등을 disk에 분산해서 저장하기 때문에 disk의 random I/O가 많이 발생합니다. 그 외에 file의 속성(inode)을 file과 분리해서 저장하고 file name을 포함한 directory도 따로 저장하기 때문에 file 하나만 write하더라도 5번 I/O가 발생합니다. 이러한 file system에서 작은 file을 write한다면 95% 이상의 bandwidth가 탐색에 사용되고 5%만이 실제 write에 사용됩니다.
   - Unix FFS에서 write할때는 async로 진행 되지만 metadata update는 sync로 진행되서 많은 작은 file을 write할때는 metadata update에 의해 성능이 저하됩니다.

   Unix FFS에서는 Write Buffer를 사용할수 없습니다.

## 3. Log-Structured File System

- Design:
  Memory를 write buffer로 사용해 data를 buffer에 write하고 가득 차면 sequential write로 disk에

write합니다.

이런 design에서 2개의 문제가 있습니다

- 1. log(buffer)에서 어떻게 data를 효율적으로 찾을 것인가?
- 2. disk에서 write할수 있는 비어있는 공간을 어떻게 찾을 것인가?

1. File Location and Reading

| Data Structure | Purpose | Location |
|---|---|---|
| Inode | Locatesblocks of file, holds protection bits, modify time, etc. | log |
| Inode map | Locates posiuon of mode in log, holds time of fast access version number. | log |
| Indirect Block | Locatesblocks of large files | log |

- Inode : file의 속성 정보와 data가 저장된 block의 주소를 최대 10개까지 저장합니다.
- Inode map : Inode는 유일한 inode number을 가지고 있고 이를 저장하는 곳이 Inode map입니다.
- Indirect Block: Inode에 저장된 block의 주소가 10개를 넘어가면 indirect block을 사용하여 data가 저장된 block의 주소를 저장합니다.

LFS에서 file을 read할때는 inode map에서 inode number를 찾고 inode에서 data가 저장된 block의 주소를 찾아서 data를 read합니다. 이때 inode map과 inode는 memory에 저장되어 있기 때문에 random I/O가 발생하지 않습니다.

2. Free Space Management: Segments
   - LFS가 disk 마지막 공간을 사용 하게 되면 file의 삭제혹은 수정에 의해 작고 많은 free space가 생기게 됩니다. 이를 해열하기 위해 2가지 방볍이 있습니다
     - Thread : free space를 찾아서 새로운 data를 write합니다(단점은 disk fragmentation이 발생합니다)
     - Copy : disk 의 마지막 block에 write할때 마다 모든 data를 앞으로 옮겨서 크고 연속된 free space를 만듭니다.(단점은 반복적인 copy가 발생하므로 cost가 큽니다)
   - 상술 된 2가지 방법의 단점을 고려해 LFS는 두 방법을 혼합한 방법을 사용합니다. Disk를 고정된 segment라는 단위로 나누어 segment의 write는 처음부터 끝까지 비우지 않고 만약 disk의 끝에 도달하면 LFS는 공간 분배, hot data의 주소 등을 고려해 같은 file의 segment를 연속적인 segment로 만듭니다.
   - Segment도 충분한 크기를 가지고 있어야 write할때 seek time의 비중이 작아집니다.
3. Segment Cleaning Mechanism

- Segment를 memory에 읽어 무효화된 data를 제거혹은 수정하고 유효화된 data들만 segmet로 disk에 다시 write 합니다.
- 이 과정중 읽어온 segment가 수정되므로 size가 작아지게 될수 있습니다, 이때는 다른 segment과 합쳐서 size가 맞은 segment를 만듭니다, 만약 size가 작다면 LFS에서는 공백data를 채워서 segment의 크기를 맞춥니다.
- Segment Summary : segment가 가지고 있는 data와 block의 주소를 저장되 있습니다. 이를 사용해 cleaning 할때 무효화 된 data를 제거 합니다.
- Inode map and Version Number : Inode map 는 Inode 의 주소와 version number를 저장되 있습니다. Cleaning 할때는 FS가 data block의 inode 와 version number을 검사하여 data block가 최신인지 어떤 file에 속하고 사용되 있는지를 확인합니다. 무효화 된 data block은 segment에서 제거합니다. LFS에서는 같은 file이지만 다른 version인 data가 disk의 block에 분산되 있을수 있습니다, 이때는 version number를 사용해 최신 data를 찾습니다.

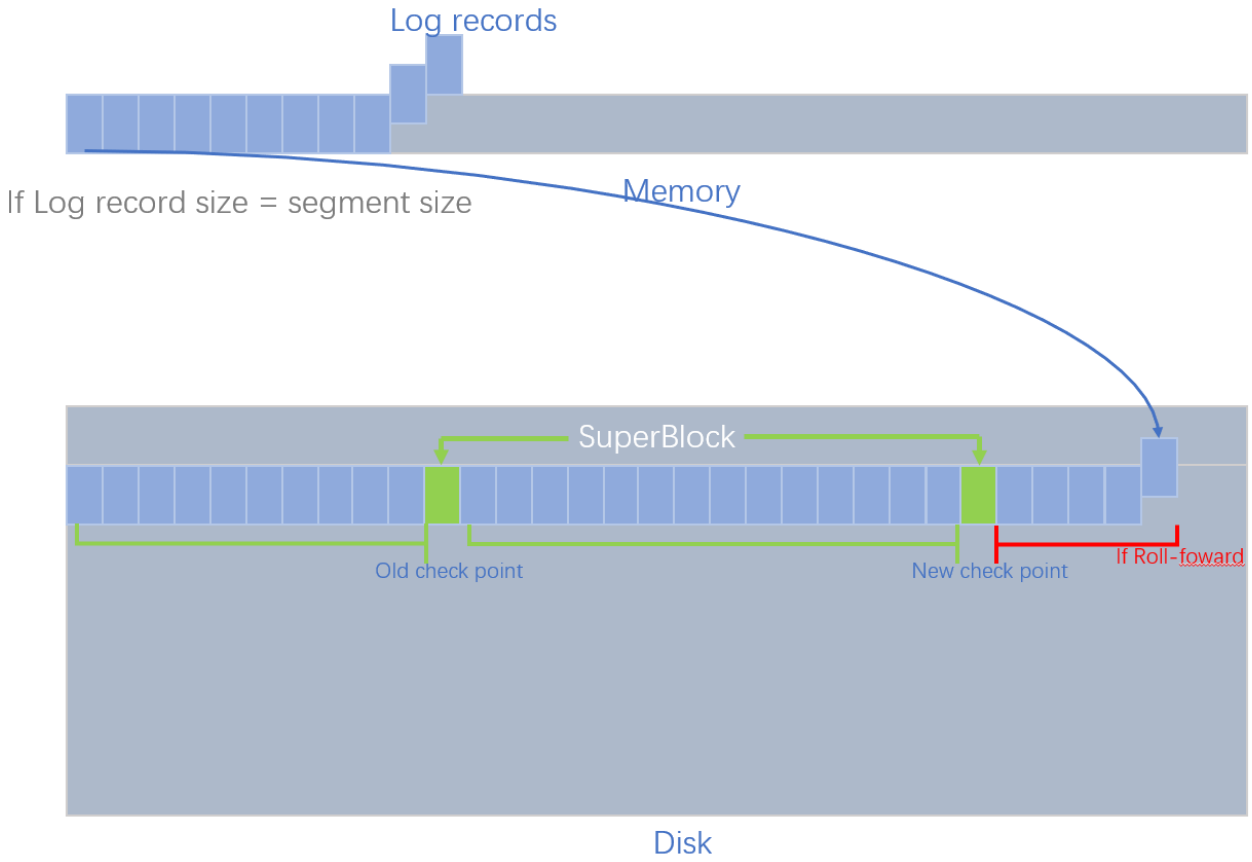
4. Segment Cleaning Policies
    1. Cleaned Segments 가 50개 이하 10개 이상이면 cleaning을 시작하고, 50개이상 100개 이하이면 cleaning을 중지하는것이 논문에서 제안하는 방법입니다.(cleaning의 trigger는 성능에 영향을 미치지 않습니다)
    2. 어떤 Segment가 Cleaning 될지는 segment의 공간 사용률(유효data 비율)을 기준으로 합니다.

$$
\begin{aligned}
write\ cost &= \frac{total\ bytes\ read\ and\ written}{new\ data\ written} \\
&= \frac{read\ segs\ + write\ live + write\ new}{new\ data\ written} \\
&= \frac{N + N * u + N(1-u)}{N * (1-u)} = \frac{2}{1-u}
\end{aligned}
$$

    u = segment의 이용율 [0~1), 식으로 보면 결론은 u가 작으면 write cost가 작아집니다.
    3. Segment를 cleaning 후 다시 write하려면 size를 맞추기 위해 다른 segment와 합쳐야 할수 있습니다. 하지만 합치는거에도 policie가 필요할수도 있습니다. LFS논문중에는 명확한 policy를 제시하지 않았지만 2가지 방법을 제시 했습니다
        1. 같은 목록으로된 data를 segment에 합칩니다.
        2. Age sort로 data의 수정시간 기준으로 segment를 합칩니다.

# 4. CRASH RECOVERY

- LFS의 file저장방식:




  - Memory를 write buffer로 사용해, data를 buffer에 저장할때는 log record방식으로 하나하나식 저장합니다. Log record에서는 data, metadata, 목록, inode map의 변경사항등 이 존재합니다.
  - log record가 disk에 저장 할때는 segment size를 고려해 size가 맞게 저장됩니다, 이때 inode map의 변경사항은 check point 생성에 사용됩니다.
  - disk에는 check point가 new(active check point), old(shadow check point) 로 2개 존재합니다.
    - Segment가 write할때 마다 log record의 inode map 변경사항을 같이 segment에 저장합니다
    - System의 부하가 낮을때 새로운 check point를 생성합니다. 새로운 check point 는 기존의 check point가 기록한 마지막 segment까지 의 정보를 가지고 그 이후의 segment의 metadata를 참조해 check point를 disk에 있는 segment(log)의 마지막 위치에 생성합니다.
    - Active Check point가 신뢰성이 있다고 판단되면 shadow check point를 삭제하며 새로운 segment가 write할때 다시 사로운 active check point를 생성합니다.
    - Check point 의 신뢰성은 segment가 write 완성되고 inode map의 모든 inode일치성을 확인하면 신뢰성이 있다고 판단합니다.
    - Check point 신뢰성 확인은 주기적으로 수행되거나 file system이 disk의 지정한 공간을 사용하면 수행됩니다.
- Crash Recovery

- Recovery 는 신뢰성이 있다는 check point에서 metadata를 읽어 Roll-Forward 를 수행합니다. 이작업은 check point가 기록한 마지막 segment후의 새로 생성된 segment의 이용율을 0으로 판단하지만 roll-forward를 수행할때는 새로운 segment를 검사해서 data가 유효하다면 새로운 check point에 기록합니다.