

**What is a DAX of Ext file system**

January 2020  
2017211795 컴퓨터 공학과 안성필

**목차**

[**1 DAX 4**](#_Toc38615852)

[**1.1 DAX란? 4**](#_Toc38615853)

[**1.2 DAX 생성배경 4**](#_Toc38615854)

[**1.3 DAX의 원리 4**](#_Toc38615855)

[**2 주의사항 5**](#_Toc38615856)

[**2.1 패키지 설치 방법 5**](#_Toc38615857)

[**2.2 Filebench 파일 링크 5**](#_Toc38615858)

[**2.3 실습환경 5**](#_Toc38615859)

[**3 사전설정 6**](#_Toc38615860)

[**3.1 DAX 사전설정[메모리 할당과정] 6**](#_Toc38615861)

[**3.2 SSD 사전설정 7**](#_Toc38615862)

[**3.3 마운트 8**](#_Toc38615863)

[**4 벤치마크 프로그램 설치 8**](#_Toc38615864)

[**4.1 FIO 설치 및 기본 설정 8**](#_Toc38615865)

[**4.2 FILEBENCH 설치 및 기본 설정 8**](#_Toc38615866)

[**5 기본 실험 9**](#_Toc38615867)

[**5.1 기본 실험(FIO) 9**](#_Toc38615868)

[**5.2 기본 실험(Filebench) 11**](#_Toc38615869)

[**6 실험 결과 12**](#_Toc38615870)

[**6.1 결과(FIO-DAX ON/OFF) 12**](#_Toc38615871)

[**6.2 결과(Filebench-DAX ON/OFF option: 4k이외 default와 동일) 13**](#_Toc38615872)

[**7 RocksDB 설치 17**](#_Toc38615873)

[**7.1 필요 파일 설치 17**](#_Toc38615874)

[**7.2 RocksDB 실행 지원 프로그램 설치 17**](#_Toc38615875)

[**7.3 RocksDB 컴파일(설치) 17**](#_Toc38615876)

[**7.4 RocksDB 실행 19**](#_Toc38615877)

[**8 YCSB 설치 20**](#_Toc38615878)

[**8.1 필요 파일 설치 20**](#_Toc38615879)

[**8.2 YCSB 실행 20**](#_Toc38615880)

[**9 YCSB를 통한 RocksDB 벤치마크 비교분석 21**](#_Toc38615881)

[**9.2 workloadA (SSD와 DAX 비교분석) 읽기 50% 업데이트 50% 21**](#_Toc38615882)

[**9.3 workloadB (SSD와 DAX 비교분석) 읽기 95% 업데이트 5% 22**](#_Toc38615883)

[**9.4 workloadC (SSD와 DAX 비교분석) 읽기 100% 22**](#_Toc38615884)

[**9.5 workloadD (SSD와 DAX 비교분석) 읽기 95% 쓰기5% 23**](#_Toc38615885)

[**9.6 workloadE (SSD와 DAX 비교분석) 읽기 95% 쓰기 5% 23**](#_Toc38615886)

[**9.7 workloadF (SSD와 DAX 비교분석) 읽기 – 쓰기 – 수정 순차실행 24**](#_Toc38615887)

[**9.8 비교분석 총 결과 24**](#_Toc38615888)

[**9.9 재확인 및 재실험 24**](#_Toc38615889)

[**9.10 재실험 결과(WorkloadA) 읽기 50% 업데이트 50% 30**](#_Toc38615890)

[**9.11 추가실험 31**](#_Toc38615891)

[**9.12 추가 결과(WorkloadA) 33**](#_Toc38615892)

[**10 참고자료 34**](#_Toc38615893)

[**10.1 사이트 34**](#_Toc38615894)

# **1 DAX**

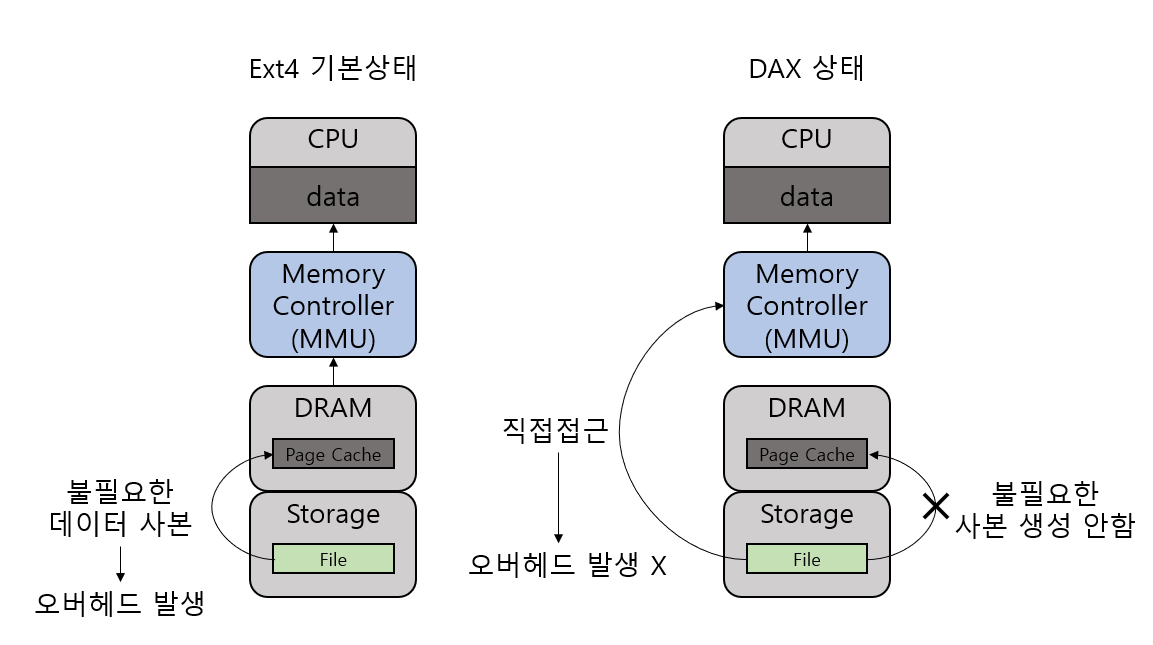
## 1.1 DAX란?

DAX는 Direct Access eXtends의 줄임말이며  
 파일 시스템에 옵션을 통해 실행이 가능합니다.

## 1.2 DAX 생성배경

일반적인 운영체제에서는 디바이스로 데이터를 불러올 때  
 필수적으로 페이지 캐시를 거쳐서 올라가게 됩니다.  
 즉, 스토리지의 파일 DRAM의 **페이지 캐시** 데이터로 올라옵니다.  
 하지만 이는 페이지 캐시의 불필요한 사본을 만들게 되고  
 이는 곧 시간적 오버헤드를 발생시키게 됩니다.  
 그렇기에 리눅스에서 DAX를 통해 불필요한 사본을 복사하는 행위를  
 없애고(혹은 무시하고) 파일을 불러오는 기술을 도입하였고  
 그 기술이 **DAX**입니다.

## 1.3 DAX의 원리

DAX를 통해 파일을 불러오게 되면 앞서 말했듯이 기존의 운영체제는  
 페이지 캐시를 거치면서 불필요한 사본을 만들고 오기 때문에  
 그만큼 속도면에서 오버헤드가 발생합니다.  
 그렇기에 굳이 페이지 캐시를 거쳐서 불필요한 사본을 생성하지 않고  
 직접적으로 접근하여 파일을 불러오는 것이 바로 DAX의 원리입니다.  
   
 그림 0 DAX의 원리(Ext4 기본상태와 Ext4에서 DAX를 ON한 상태)

# **2 주의사항**

## **2.1 패키지 설치 방법**

필요 패키지는 “sudo apt-get install 설치패키지”를 입력해줍니다.  
 ex) **sudo apt-get install make**  
 이는 명령어가 제대로 실행되지 않았을 때도 포함합니다.  
 그림 1 패키지가 존재하지 않는 경우

## **2.2 Filebench 파일 링크**

본 실험에서는 최신버전의 Filebench 프로그램이 정상적으로  
 작동이 되지 않아서 구 버전인 1.4.9.1의 Filebench를 사용했음을 미리  
 알려드립니다.  
 [**https://sourceforge.net/projects/filebench/files/1.4.9.1/**](https://sourceforge.net/projects/filebench/files/1.4.9.1/) 참고 부분: 2.2 Filebench

## **2.3 실습환경**

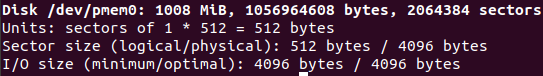
실습환경은 아래와 같습니다.

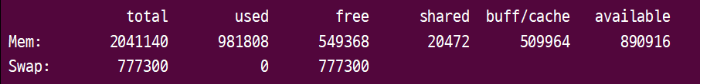
|  |  |
| --- | --- |
| CPU | Intel i7 – 6700 ( 4 core 8 thread ) |
| RAM | Samsung DDR4 8GB |
| Storage | Samsung SSD 860 PRO 256GB |
| Linux version | Ubuntu 18.04.3 LTS |
| Kernel version | 5.0.0-36-generic |

표 1 실습환경

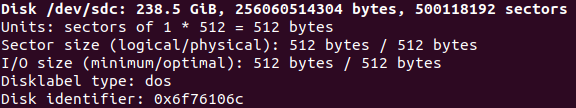
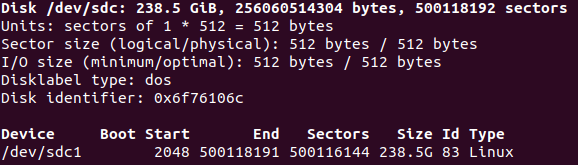
# **3 사전설정**

## **3.1 DAX 사전설정[메모리 할당과정]**

DAX를 사용하려면 메모리를 디스크처럼 할당해주어야 합니다.  
 메모리 할당 이전에 사용 가능한 메모리를 확인해야 됩니다.  
 ○ 터미널에 **free 입력**  
  
  
 그림 2 메모리 확인(free 명령어)  
 사용 가능한 메모리가 확인 되었으면 이제 직접 할당해주어야 됩니다.  
 ○ 아무 폴더를 엽니다.  
 ○ 좌측의 **other탭을 클릭**  
 ○ **경로 /etc/default/ 폴더 안에 grub 파일 열기**  
 ○ 이때 파일은 기본적인 텍스트 열기 프로그램을 사용  
 ○ GRUB\_CMDLINE\_LINUX=””를  
 **GRUB\_CMDLINE\_LINUX=”memmap=4G!4G” 로 변경**  
 이는 **4G의 물리적 메모리 공간 이후 4G의 새로운 매핑을 생성한다는 의미**입니다.  
 **단, 자신의 사용가능한 메모리 이상의 메모리를 매핑할 시  
 정상적인 매핑이 이루어지지 않고 리눅스에 문제가 생기거나 일부 매핑에 실패합니다.**  
 그러므로 위에서 확인한 사용가능한 메모리 범위 내에서 해주시기 바랍니다.  
 **ex) 12G!4G는 16G(총 메모리) 메모리 중 4G 이후 12G를 매핑함(12G 할당)** ○ 이후 저장 후 터미널에 **sudo update-grub2 입력** ○ **컴퓨터 재부팅** 이후 터미널을 다시 켠 후 **sudo fdisk -l** 입력  
   
 그림 3 pmem 공간 할당 확인  
 ○ 이후 터미널에 **sudo mkfs.ext4 /dev/pmem0** 입력  
 ○ **Writing superblocks and filesystem accounting information: done** 확인  
 만약 이 과정에서 문제가 생겼을 경우 처음부터 다시 반복합니다.



## **3.2 SSD 사전설정**

SSD 또한 ext4 포맷으로 실험을 진행해야 하기에 파티션을 생성해줍니다.  
 ○ 터미널에 **sudo fdisk -l** 입력  
 리눅스에서는 기본적으로 디스크의 구분을 /dev/sd? 로 구분합니다.  
 자신의 드라이브가 첫번째면 /dev/sda 두번째면 /dev/sdb입니다.  
 이에 맞춰 자신의 드라이브를 찾습니다.  
   
 그림 4 디스크 확인(fdisk -l 명령어)  
 ○ 터미널에 **sudo fdisk** 디스크 입력  
 ex) sudo fdisk /dev/sdc  
 ○ **n, p**입력 이후 **Command (m for help):** 문구가 나올 때까지 **Enter**입력  
 만약 위 문구가 나오지 않았을 경우 다시 반복  
 ○ w 입력  
 ○ 터미널에 **sudo fdisk -l** 입력  
 이후 해당 스토리지에 파티션이 추가되었는지 확인 후 없으면 다시 반복  
   
 그림 5 파티션 확인(fdisk -l 명령어)

## **3.3 마운트**

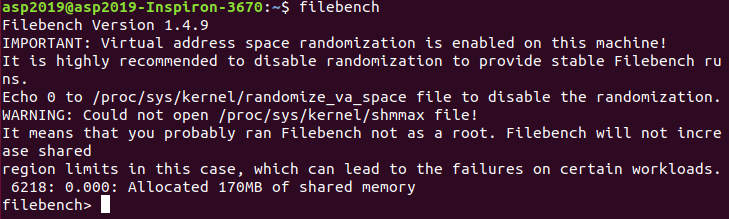
○ 터미널에 sudo mount /dev/파티션 마운트위치 입력  
 ex) **sudo mount /dev/sdc1 Desktop/ssd**  
 ○ 터미널에 sudo -o dax mount /dev/파티션 마운트위치 입력  
 ex) **sudo -o dax mount /dev/pmem0 Desktop/ramdisk** ○ 이후 터미널에 **sudo mount** 입력  
 그림 6과 같이 나오면 정상적으로 연결된 것입니다.  
   
 그림 6 각 마운트 완료 모습

# **4 벤치마크 프로그램 설치**

## **4.1 FIO 설치 및 기본 설정**

○ 터미널에 **sudo apt-get install fio** 입력  
 ○ **Do you want to continue** 문구에 **Y** 입력  
 ○ 이후 기다린 후 터미널에 **fio -v** 입력  
 fio-x.x 등의 버전이 나온다면 정상적으로 설치된 것입니다.  
 **ex) fio-3.1**  
   
 그림 7 fio 버전 확인 (fio -v 명령어)

## **4.2 FILEBENCH 설치 및 기본 설정**

**Filebench 설치 시 Command not found 와 같은 문구가 발생 시** [이곳을 클릭해 주세요.](#_2.1_패키지_설치)  
 위 실험에서는 최신 버전의 오류로 인해  
 구 버전인 1.4.9 버전을 사용하였습니다.  
 <https://github.com/filebench/filebench>  
 ○ 링크 클릭 후 **Clone or download**를 클릭 후 ZIP 파일 다운  
 ○ 해당 ZIP을 압축풀기  
 ○ 압축 풀기 된 폴더를 연 후 오른쪽 클릭  
 ○ Open in terminal 을 통해 터미널 실행  
 ○ **sudo ./configure** 입력  
 ○ **sudo make** 입력  
 ○ **sudo make install** 입력  
 ○ **filebench** 입력  
 **filebench>** 가 나온다면 정상 설치 된 것이며 아닐 시 다시 반복  
   
 그림 8 filebench 실행 모습

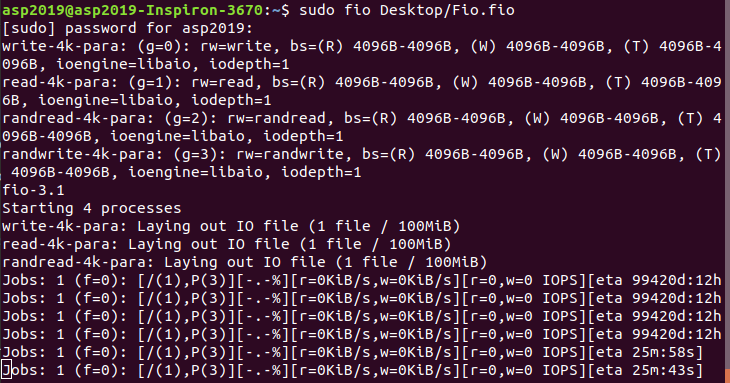
# **5 기본 실험**

## **5.1 기본 실험(FIO)**

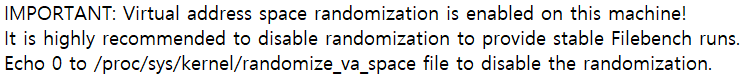
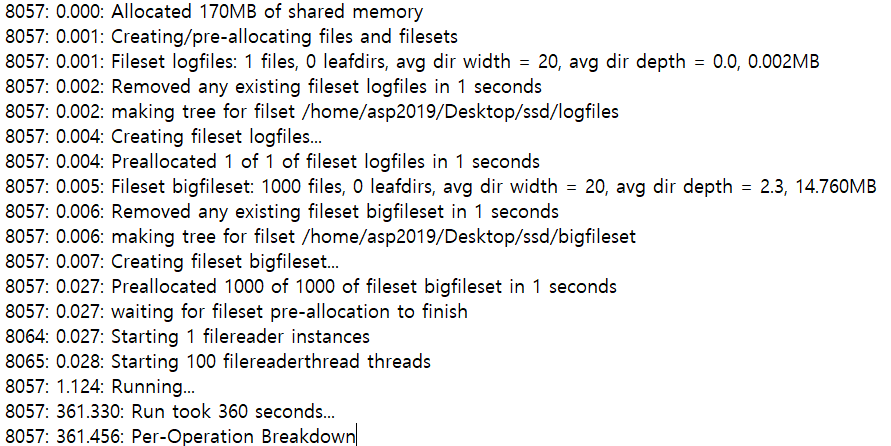
○ FIO 설정파일 생성(설정파일 참고)

|  |
| --- |
| [global] #전역변수  direct=1 #1이면 다이렉트, 0이면 버퍼  ramp\_time=30 #job 간의 여유시간  size=100MB #job 별 파일의 크기  iodepth=1 #큐의 깊이 1이상이어도 상관없지만 1이상일 경우 libaio에서 효과가 없을 수 있음  runtime=60 #실행되는 시간  time\_based #runtime 시간만큼 I/O 실행  group\_reporting #job별이 아닌 그룹별 리포팅  norandommap #랜덤 워크로드 시 I/O 위치를 과거 위치를 고려하지 않음  ioengine=libaio #I/O 방식 설정  bs=4k #blocksize이며 2k, 4k, 8k 등 2배수 사용  stonewall #각 파라미터가 한개의 파라미터가 끝나기 전까지는 다른 파라미터 작업 안함  [write-4k-para] #지역변수(이름은 변경되어도 상관없음)  #복수지정 가능 및 파일크기  **filename=/home/asp2019/Desktop/ssd/wtest** #FIO 실험을 할 폴더  rw=write #파일을 쓰는 방식 설정(write, read, randread, randwrite)  [read-4k-para]  **filename=/home/asp2019/Desktop/ssd/rtest**  rw=read  [randwrite-4k-para]  **filename=/home/asp2019/Desktop/ssd/rdwtest**  rw=randwrite  [randread-4k-para]  **filename=/home/asp2019/Desktop/ssd/rdrtest**  rw=randread |
| 설정파일(FIO.fio) |

* 터미널에 sudo fio 설정파일이름 입력  
  ex) **sudo fio FIOFILE**  
  **sudo fio FIOFILE >> output.txt** 등을 통해 결과파일 한번에 저장가능  
  그림 9와 같이 나오지 않을 시 다시 반복

  
 그림 9 FIO 정상 구동 화면

## **5.2 기본 실험(Filebench)**

Filebench는 설치 시 사용하였던 폴더의 **Workload라는 폴더** 안에 설정파일이 있습니다.  
 그 중 해당 실험에서는 webserver.f, fileserver.f, varmail.f 만 했습니다.  
 ○ 설치 시 사용한 폴더의 Workload 폴더 열기  
 ○ 해당 폴더에 오른쪽 클릭 후 Open in terminal 클릭  
 ○ 터미널에 sudo filebench -f 설정파일이름.f  
 ex) **sudo filebench -f webserver.f** 이후 리눅스의 디스크 랜덤 할당 옵션 관련 오류가 발생합니다.  
 (발생하지 않는다면 그대로 진행합니다.)  
 ○ **su** 입력 후 비밀번호 입력하여 관리자 권한 획득  
 ○ **sudo echo 0 to /proc/sys/kernel/randomize\_va\_space** 입력  
 이후 정상적으로 실행됩니다.  
 만약 그림 11과 같이 정상적으로 실행되지 않는다면 다시 반복  
  
   
 그림 10 디스크 랜덤 할당 옵션 오류  
  
   
 그림 11 정상실행 화면

# **6 실험 결과**

# **6.1 결과(FIO-DAX ON/OFF)** 앞서 언급했듯이 DAX에 대한 실험을 통해 SSD와 DAX간의 관계분석을 해보도록 하였습니다. 메인 메모리 16GB중 12GB를 영구적 메모리 사용을 통해 DAX로 할당하였고 파일시스템은 Ext4로 SSD 또한 같은 파일시스템으로 실험을 진행하였습니다.

첫번째 실험은 SSD와 영구적 메모리 사용을 통해 DAX를 켠 메모리와의 성능 분석입니다.  
DAX와 SSD간의 Ext4 파일시스템의 분석 기준이 필요하여 FIO를 통해 기본상태를 알아봤습니다.  
  
그 결과 FIO에서는 순차쓰기, 순차읽기, 임의쓰기, 임의읽기 모두 DAX가 ON인 상태에서와 OFF인 상태에서의 차이가 약 10배에서 크게는 28배까지 차이가 났습니다.

6.2 결과(Filebench-DAX ON/OFF option: 4k이외 default와 동일)  
이번에는 FIO가 아닌 Filebench 벤치마킹 프로그램을 통해서 기본상태의  
SSD와 DAX를 비교하여 분석해보았습니다.

그 결과 Filebench에서는 fileserver, webserver, varmail 세가지를 사용하였으며, 각각의 값들은  
fileserver에서는 20정도(**약 1.1배**), webserver에서는 비슷하며, varmail에서는 150정도(**약 12배**)  
**값의 변화를 확인**할 수 있었습니다.

**6.3 결과(FIO-DAX ON/OFF** option: numjobs=16, size=10MB 이외 모두 동일**)**이번에는 FIO를 통해 DAX의 성능을 극대화 하기 위해 어떠한 옵션이 필요할 지 찾다가  
동일한 옵션을 주었을 때 numjob(쓰레드 수)가 커짐에 따라 전체적인 성능 변화가 일어난 것을  
볼 수 있었습니다.  
확인 결과 FIO에서 다중 쓰레드를 통해 작업을 할 경우 메모리에 직접적 연결을 하는 DAX는  
CPU에 전달되는 과정에서 시간적 오버헤드를 발생하는 SSD보다 변화 폭이 더 컸습니다.  
  
해당 실험에서는 일반적인 RAM 상태와 SSD 그리고 마지막으로 DAX를 살펴보았습니다.  
RAM의 경우 DAX를 켰을 때와 아닐 때의 비교를 통해 오버헤드 발생이 어느 것이 더 심한 지  
알아보기 위해 추가하였습니다.  
그리고 위에서 언급했듯이 numjob(쓰레드 수)을 통해 다중 쓰레드에서  
RAM(DAX OFF)과 SSD 그리고 DAX가 전체적으로 성능이 증가했음을 알 수 있었습니다.  
순차쓰기, 순차읽기, 임의쓰기, 임의읽기 각각의 분야에서 증가를 하였지만 단일 쓰레드였을 때  
DAX(1996), RAM(1695), SSD(160)인 순차쓰기 성능에서는 DAX가 RAM보다 약 300차이가  
났고 RAM은 SSD보다 약 1530차이가 났습니다.  
이후 단일 쓰레드를 numjob을 통해 다중 쓰레드로 분석을 하였을 때는  
DAX(3226), RAM(2316), SSD(392)인 순차쓰기 성능에서는 DAX가 RAM보다 약 910차이가  
났고 RAM은 SSD보다 약 1920차이가 났습니다.  
이 중에서 가장 큰 차이를 볼 수 있는 DAX와 SSD차이에서는 **단일 쓰레드는 약 1830** **다중 쓰레드는 약 2830**이라는 차이를 보이며   
numjob을 통한 다중 쓰레드로 기존대비 약 150%의 성능 향상이 일어났습니다.   
이는 DAX를 사용하였을 때 스토리지에 있는 파일을 CPU로 전달할 때 DRAM에 있는  
페이지 캐시를 거치지 않고 가기 때문에 다중 쓰레드를 사용하였을 때 CPU로 전달되는  
파일이 많아지면서 더욱 큰 성능을 보여준 것으로 보입니다.  
**이는 앞으로 쓰레드의 수를 증가시킴에 따라 더 많은 차이를 볼 수 있음을 기대할 수 있었습니다.  
그렇기에 DAX를 최대한 성능을 높일 수 있는 방법 중 하나라고 생각합니다.  
추가적으로 BandWidth에서도 확인을 해보았습니다.**

BandWidth 역시 기존에 차이가 있었으며, 위에서 같은 방식으로 살펴볼 때  
단일 쓰레드일 때 순차쓰기의 DAX와 SSD의 차이는 약 1920 이며, numbjob을 통한  
다중 쓰레드일 때는 DAX와 SSD의 차이가 약 3030으로 기존대비 약 50%가량  
증가하였음을 알 수 있었습니다.  
이 또한 numjob을 통해 DAX의 성능을 극대화 할 방법이라는 근거로 제시된다고 생각합니다.

7 RocksDB 설치

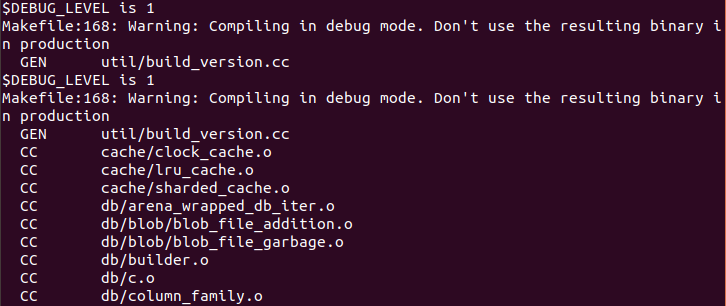
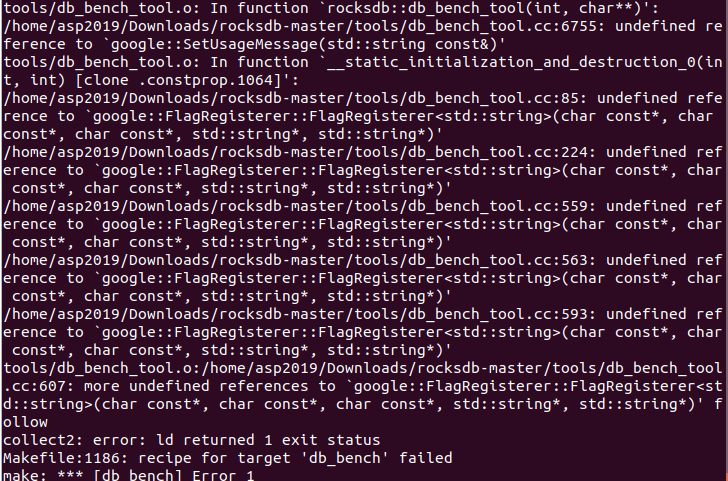
## **7.1 필요 파일 설치**

RocksDB는 C 컴파일을 사용한 뒤 사용이 가능합니다.  
그렇기에 C (gcc)와 C++ (g++)컴파일러가 요구됩니다.  
○ sudo apt-get install gcc   
○ sudo apt-get install g++

## **7.2 RocksDB 실행 지원 프로그램 설치**

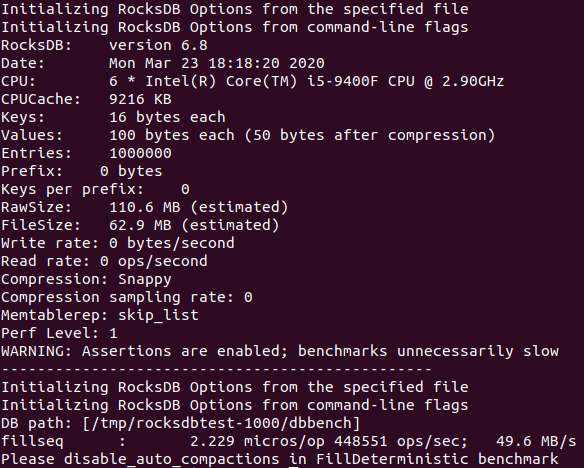
RocksDB는 지원 프로그램이 없이는 실행되지 않습니다.  
해당 벤치마크 프로그램을 실행하기 위해 몇 가지 지원프로그램을 설치합니다.  
○ sudo apt-get install libgflags-dev **- libgflags 설치**(지원 프로그램중 가장 중요하니 꼭 정상적으로 설치요구)  
(만약 정상적인 설치가 되지 않는다면 아래 주소를 확인해주세요)  
[<http://askubuntu.com/questions/312173/installing-gflags-12-04>]  
이후 터미널에 계속해서 명령어를 입력해주세요.  
○ sudo apt-get install libsnappy-dev **- libsnappy-dev 설치**  
○ sudo apt-get install zlib1g-dev **- zlib1g 설치**○ sudo apt-get install libbz2-dev **- libbz2-dev 설치**  
○ sudo apt-get install liblz4-dev **- liblz4-dev 설치**○ sudo apt-get install libzstd-dev **- libzstd-dev 설치**  
설치 도중 오류가 떴다면 다시 설치합니다.

## **7.3 RocksDB 컴파일(설치)**

터미널을 실행한 후 **cd** 명령어를 통해원하는 폴더로 이동합니다.  
이후 RocksDB를 다운로드해줍니다.  
○ git clone https://github.com/facebook/rocksdb.git  
그러면 처음에 이동한 폴더안에 RocksDB 폴더가 생성됩니다.  
해당 폴더 안에서 오른쪽 클릭 후 터미널을 열어줍니다.  
해당 터미널에서 명령어 실행  
○ sudo make  
추가적으로 RocksDB의 설명서에는 make static\_lib를 추천하나  
이는 RocksDB가 실행이 안됩니다.  
그로 인해 make를 하였으나 실행하는 것에는 문제가 없습니다.  
  
그림 12 RocksDB의 컴파일 과정  
  
그림 13 RocksDB 컴파일 중 오류발생  
그림 13과 같이 오류가 발생 시 gcc와 g++이 정상적으로 설치가 되지 않거나  
버전이 낮은 상황이므로 번호 7.1로 돌아가서 다시 설치하면 됩니다.

## **7.4 RocksDB 실행**

컴파일 이후 정상적으로 완료되었다면  
○ sudo ./db\_bench  
명령어를 입력해줍니다.

  
그림 14 정상적인 RocksDB 결과  
명령어가 정상적으로 실행된 경우 그림 14와 같이 정상적인 결과가 나오게 됩니다.

8 YCSB 설치

## **8.1 필요 파일 설치** 앞서 본 RocksDB와는 다르게 YCSB는 자바를 통해 사용됩니다. 그렇기에 필요한 프로그램으로 자바와 메이븐이 존재하며 이 두개의 프로그램을 설치하여야 정상적인 작동이 됩니다. 터미널을 열어서 명령어를 입력해줍니다. ○ sudo apt-get install maven 이후 문제가 없이 정상적으로 설치가 되었다면 확인을 해봐야합니다. ○ sudo java --version 그림 15 자바 버전확인 그림 15와 같이 오류가 없이 나온다면 정상적으로 설치된 것입니다.

## **8.2 YCSB 실행** 실행 방법에 관해서는 여러가지 방법이 존재합니다. 폴더 안에 RocksDB가 존재하기에 해당방법을 통해서 YCSB를 실행해봤습니다. (예시문의 경우 RocksDB의 ReadMe 파일을 통해 있던 예시문을 사용했습니다) ○ sudo ./bin/ycsb load rocksdb -s -P workloads/workloada -p rocksdb.dir=/tmp/ycsb-rocksdb-data 그림 16 YCSB 실행 확인 그림 16과 같이 결과가 나온다면 YCSB 또한 정상적으로 작동하는 것입니다.

9 YCSB를 통한 RocksDB 벤치마크 비교분석

**9.1 사전지식**YCSB는 자체적 실행이 아닌 RocksDB Hbase 등의 벤치마크를 통해 같이 실행되는 형태입니다.  
그렇기에 이번 실험에서는 RocksDB를 매개체로 한 벤치마크를 하려고자 합니다.  
실험은 각각 기본적으로 구성된 workload의 a부터 f까지의 성능을 분석할 것이며 이는 각각

Workloada - 읽기50%, 업데이트50% (업데이트 중심)  
Workloadb - 읽기95%, 업데이트5% (읽기 중심)  
Workloadc - 읽기100% (오직 읽기)  
Workloadd - 읽기95%, 쓰기5% (최근 기록 중심 읽기)  
Workloade - 읽기95%, 쓰기5% (영역 스캔)  
Workloadf – 읽기 – 쓰기 – 수정 순차실행  
  
이는 workloadtemple을 통해 새로운 성능 분석이 가능한 방식의 workload를 자신이 만들 수도 있습니다.

그리고 해당 테스트의 경우 기본적으로 MinLatency(us)가 거의 동일하거나 같은 수준으로 나왔기에 테스트 결과에서는 제외하였습니다.  
  
**그리고 기준은 recordCount는 RAM용량상 4GB이내로 잡기 위해 100만을 주었고  
OperationCount는 더 정확한 분석을 위해 10억을 주고 실험에 임했습니다.  
이외에 명령어의 경우**○ sudo ./bin/ycsb load rocksdb -s -P workloads/workloada -p rocksdb.dir=/tmp/ycsb-rocksdb-data

**를 그대로 사용하였으며 해당 rocksdb.dir= 의 위치만 각 마운트 된 폴더로 해주었습니다.   
(마운트에 대한 자세한 내용은** [**3번의 사전설정**](#_3_사전설정) **토대로 했습니다)  
그리고 추가적으로 workload 폴더 안의 workload파일들에 maxexecutiontime 옵션으로 10800(분)으로 주어서 3시간 정도 실행했습니다.**

​

## **9.2 workloadA (SSD와 DAX 비교분석)** 읽기 50% 업데이트 50%

워크로드 A의 경우 Throughput이 상대적으로 차이가 났으며 약 1.04배 정도 차이가 났습니다.  
아무래도 읽기와 업데이트가 섞여 있어서 낮은 차이를 보인 것이 아닐까 생각합니다.

## **9.3 workloadB (SSD와 DAX 비교분석)** 읽기 95% 업데이트 5%

워크로드 B의 경우 읽기 95%와 업데이트 5%로써 상대적으로 읽기의 비율이 많습니다.  
그렇기에 약 1.5배정도 차이가 났으며 앞서 본 workload A 이상의 차이를 보여주었습니다.

## **9.4 workloadC (SSD와 DAX 비교분석)** 읽기 100%

워크로드 C의 경우 읽기 100퍼입니다. 오로지 읽기만 하는 부류인데도 불구하고 차이는  
1.01배로 큰 차이는 볼 수 없었습니다.

## **9.5 workloadD (SSD와 DAX 비교분석)** 읽기 95% 쓰기5%

워크로드 D의 경우 최근 기록에 대한 읽기95%와 쓰기5%를 수행합니다.  
해당 작업의 경우 1.6배가 차이가 났으며 기록에 대해서 읽기를 95%로 작업했기에  
앞서 보았던 Workload B보다 더 높은 수치를 보여준 것이 아닐까 생각합니다.

## **9.6 workloadE (SSD와 DAX 비교분석)** 읽기 95% 쓰기 5%

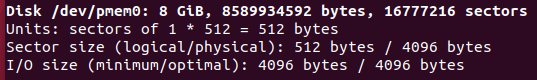
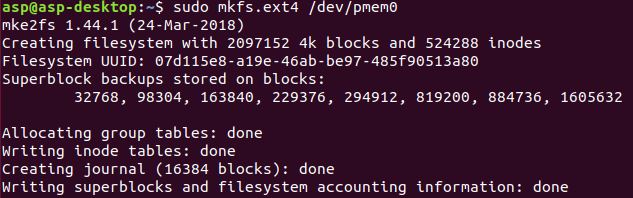
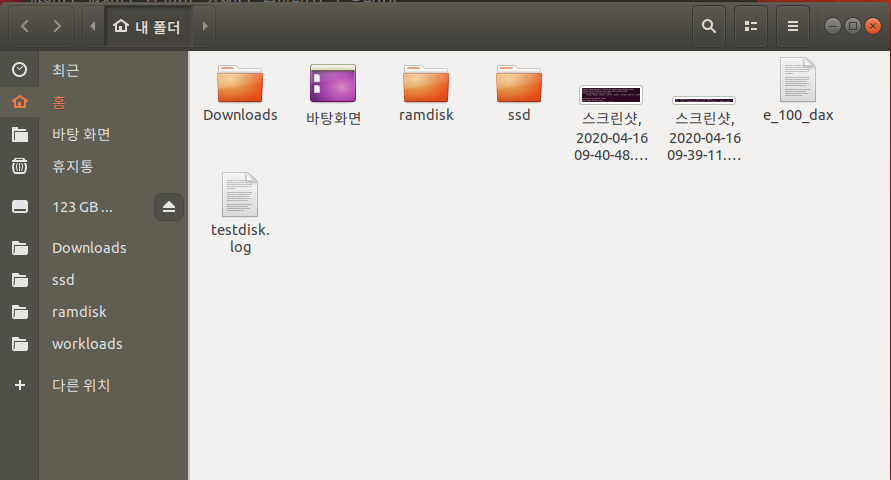
워크로드 E의 경우 스캔영역을 읽기 95%와 쓰기 5%를 수행합니다.  
워크로드 E의 경우 다른 워크로드와는 다르게 낮은 수치를 기록하였으며  
1000분의 1로 표현하였기에 같은 수치이지만  
제대로 된 수치는 약 8243(DAX), 약 8278(SSD)로 오히려 SSD가 더 높은 기록을 남겼습니다.

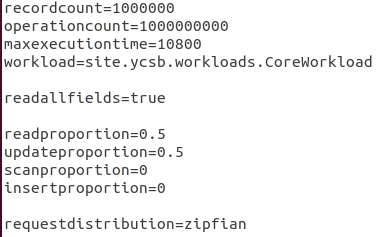
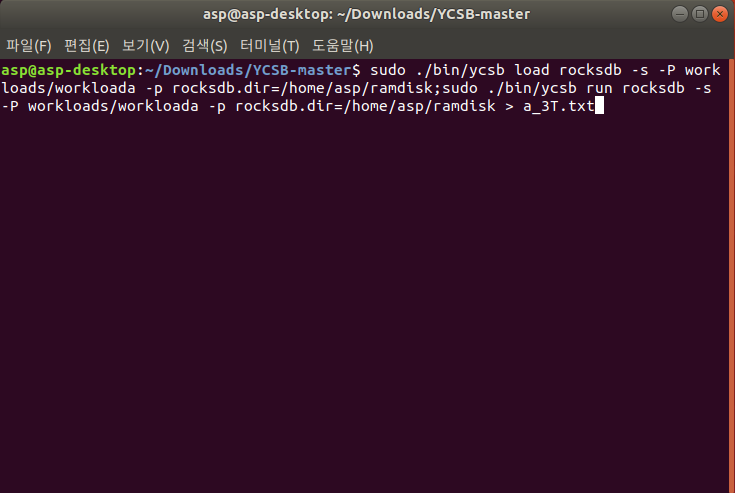
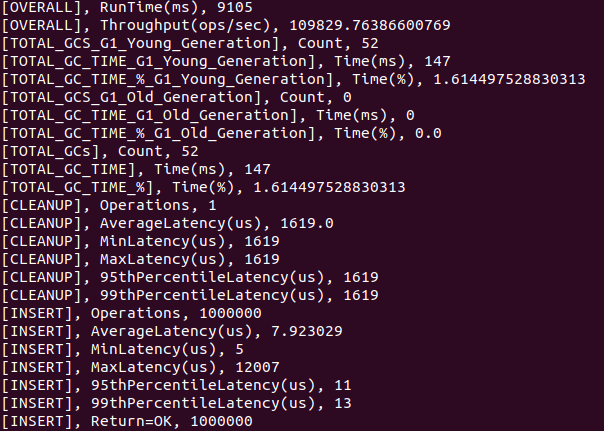
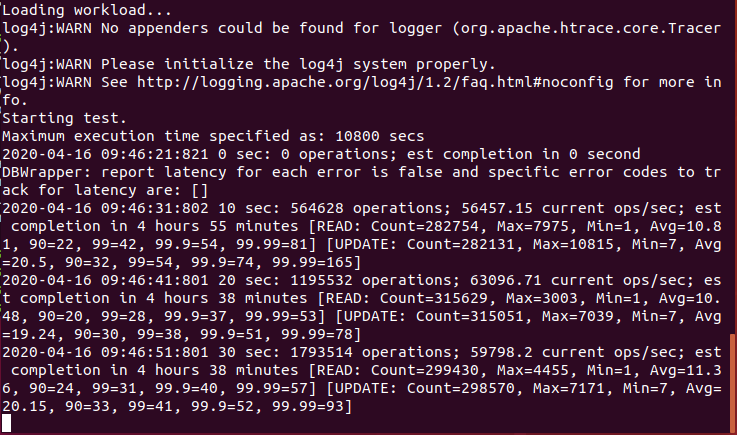
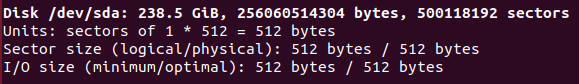
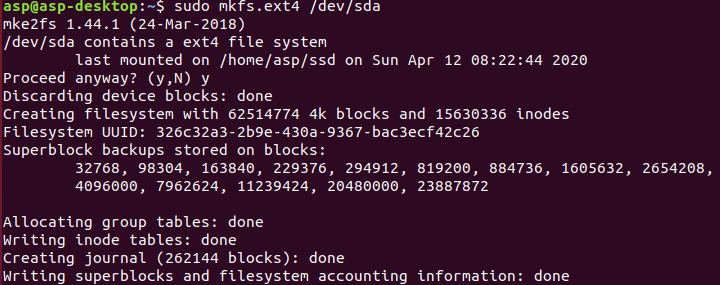
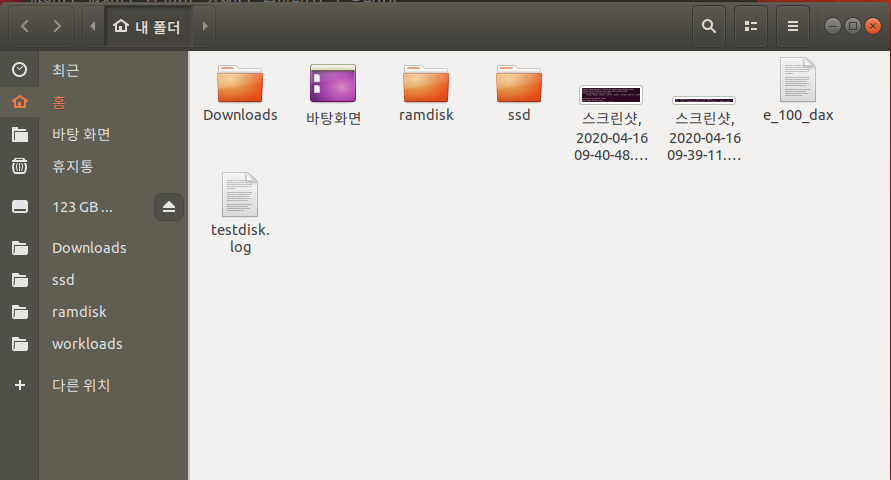
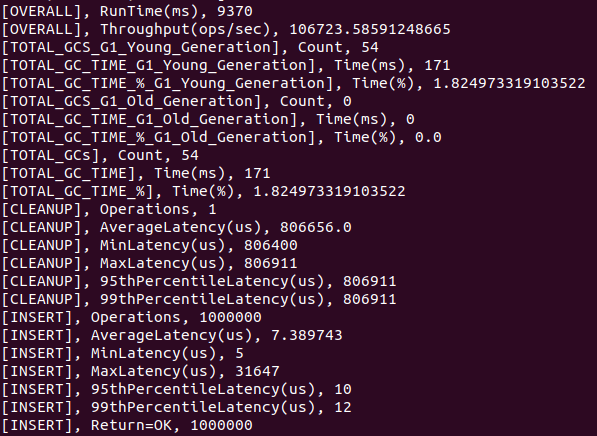
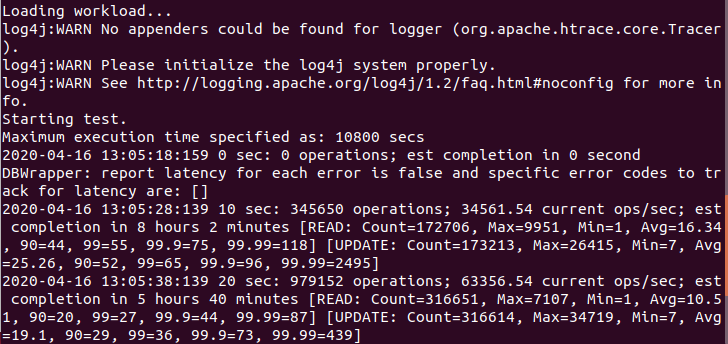
## **9.7 workloadF (SSD와 DAX 비교분석)** 읽기 – 쓰기 – 수정 순차실행

워크로드 F의 경우 읽기, 쓰기, 수정을 순차적으로 실행하는 작업을 수행합니다.  
해당 작업을 실행한 결과 DAX가 SSD에 비해 약 1.04배 차이가 나는 기록을 보여주었습니다.

## **9.8 비교분석 총 결과** 몇몇 부분에서는 큰 차이를 보여주어 나름대로의 결론이 맞다고 생각이 듭니다만 그 외의 부분에서는 낮은 차이를 보여주었기에 이해가 안가는 부분이 존재합니다. 하물며 기록이 100%인 Workload C의 경우 기존의 DAX가 캐시를 남기지 않고 직접적으로 불러오는 작업을 하는 것이기에 분명히 훨씬 더 빨라야 된다고 생각합니다만 오히려 다른 Workload에 차이가 밀리는 것을 확인할 수 있었습니다. 이 점에 대해서는 더 연구를 진행해봐야 된다고 생각합니다.

## **9.9 재확인 및 재실험** 기존에 DAX의 경우 RAM을 이용한 작업이기 때문에 기본적인 DRAM 속도와 더불어 DAX라는 기술을 사용하였기에 SSD와의 성능차이는 분명히 많이 차이가 난다고 생각했습니다. 그러나 이번 실험에서 해당 테스트의 문제가 있지 않을지 다시 한번 확인을 해봐야 된다고 생각했기에 재실험을 해봐야 된다고 생각이 들었습니다. 그렇기에 이번에는 한끝의 실수 없이 실험에 임하기 위하여 실험 과정을 체크하고 확인하며 진행하였습니다.

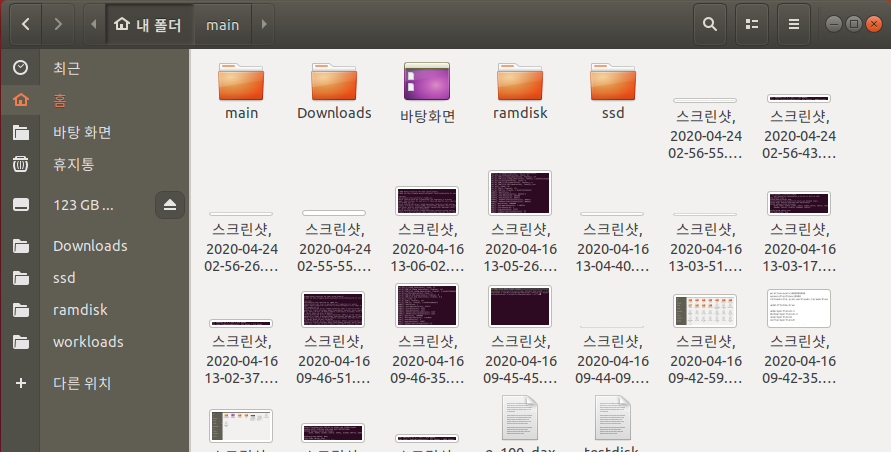
DAX를 사용하기 위해 [사전설정](#_3.1_DAX_사전설정[메모리)을 미리 해줍니다.  
메모리를 할당해주었으면 정상적으로 되었는지 확인을 해야 됩니다.  
○ sudo fdisk -l  
이후 해당하는 화면에서 Pmem0이 제대로 할당이 되었는지 확인을 해줍니다.  
  
그림 17 pmem0 메모리 할당 성공  
그림 17과 같이 pmem0이 나온다면 정상적으로 메모리 할당이 완료한 것입니다.  
이후 mkfs 명령어를 이용하여 ext4 파일시스템으로 만들고 포맷 시켜줍니다.  
○ sudo mkfs.ext4 /dev/pmem0   
그림 18 pmem0 ext4 포맷  
그림 18과 같이 pmem0을 정상적으로 ext4 파일시스템으로 포맷해주었습니다.  
이후 자신이 원하는 폴더에 마운트 할 폴더를 작성해줍니다.  
그림 19 마운트 할 폴더 생성  
그림 19와 같이 마운트 할 폴더를 생성해주었으면 이제 마운트를 해주어야 됩니다.

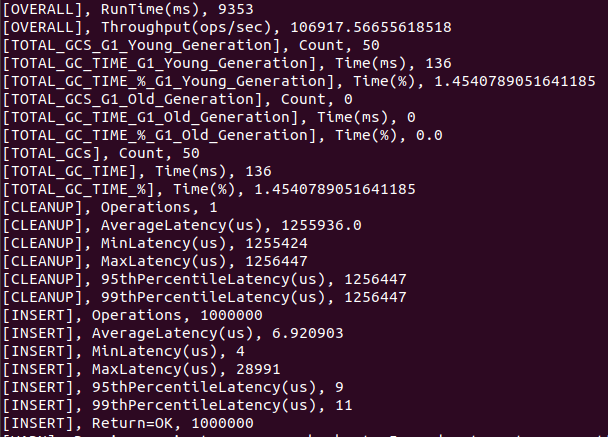
○ sudo mount -o dax /dev/pmem0 ramdisk  
이후 YCSB 폴더로 들어가서 Workload 폴더로 들어간 뒤  
Workloada를 열어서 주석처리 된 부분을 제외하고 고쳐줍니다.  
  
그림 20 Workloada 파일  
Workloada의 파일은 모든 부분은 그대로 두되 빨간색 부분인  
**recordcount=10000  
operationcount=1000000000  
maxexecutiontime=10800**  
으로 설정해줍니다.  
이 중 maxexecutiontime 옵션은 사용자가 원하는 시간대에 벤치마크가 종료되도록  
도와주는 옵션입니다. (해당 실험에서는 10800분이므로 3시간 동안 실행예정)  
이후 ycsb를 실행해줍니다.  
그림 21 YCSB 실행(DAX)  
○ sudo ./bin/ycsb load rocksdb -s -P workloads/workloada -p  
 rocksdb.dir=/home/asp/ramdisk;sudo ./binycsb run rocksdb -s -P workloads/workloada -p rocksdb.dir=/home/asp/ramdisk > a\_3T.txt  
**-s 는 10초마다 진행상황을 알려주며  
-P(대문자) 는 실행할 워크로드를 불러옵니다.  
-p(소문자) 는 rocksdb.dir=을 통해서 rocksdb에 쓰일 공간을 정해줍니다.  
해당 실험에서는 ramdisk와 ssd이므로 각 마운트 된 곳으로 설정해주었습니다.  
; 는 다중실행 명령어이며 세미콜론 이후 나오는 명령어를 또 실행해주는 역할을 합니다.  
해당 실험에서는 load의 영향은 불필요하기에 load이후 즉시 run을 해주도록 했습니다.  
> 은 결과 출력 명령어입니다. 해당 명령어 뒤에 파일이름을 쓰면 결과가 저장됩니다.**  
그림 22 load 완료(DAX)  
DAX의 load가 완료되어 나온 결과입니다.  
그림 23 run 실행(DAX)  
이후 load를 통해 생성된 파일을 run을 통해 즉시 실행하게 됩니다.  
이후 정상적으로 결과가 나와서 다음 실험을 위해 컴퓨터를 재부팅 했습니다.  
그림 24 SSD 확인  
이후 실험의 경우 앞서 했던 일들과 유사한 방식으로 이루어졌습니다.  
그림 24와 같이   
○ sudo mount -o dax /dev/sda ssd  
를 통해서 SSD가 정상적으로 인식 되어있는지 확인했습니다.  
그림 25 SSD ext4로 파일시스템 변환 및 포맷  
○ sudo mkfs.ext4 /dev/sda  
그림 25에 보시는 바와 같이 연결된 SSD 또한 DAX와 같은 Ext4 파일시스템으로 포맷  
시켜주었습니다.  
이후 SSD 또한 마운트 할 폴더를 생성시켜 주었습니다.  
그림 26 SSD 마운트 폴더 생성  
○ sudo mount /dev/sda ssd해당 명령어를 통하여 사진 26에서 생성한 SSD 폴더에 연결된 SSD를 마운트 시켜주었습니다.  
이후 실험은 동일하게 진행하되 폴더는 마운트 된 곳으로 해줍니다.  
****그림 27 YCSB 실행(SSD)○ sudo ./bin/ycsb load rocksdb -s -P workloads/workloada -p  
 rocksdb.dir=/home/asp/ssd;sudo ./binycsb run rocksdb -s -P workloads/workloada -p rocksdb.dir=/home/asp/ssd > a\_3T\_ssd.txt  
위 실험에서는 rocksdb.dir=를 기존 ramdisk에서 SSD가 마운트 된 ssd 폴더로  
변경했습니다.  
그림 28 load 완료(SSD)  
SSD의 load가 완료되어 나온 결과입니다.  
  
  
  
  
  
  
그림 29 run 실행(SSD)  
이전 실험과 동일하게 load가 완료되면 즉시 run이 실행되도록 수행하였습니다.

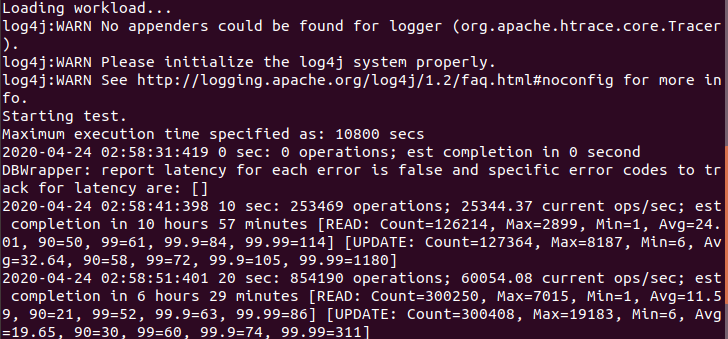
## **9.10 재실험 결과(WorkloadA)** 읽기 50% 업데이트 50%결과는 약 3퍼의 차이가 나는 것으로 확인이 되었으며 앞서 실험했던 것에 비교하였을 때 4퍼의 차이보다 1퍼 낮은 값이며 거의 유사한 값이 나왔습니다. DAX라는 점, 그리고 RAM을 사용한 것으로 결과가 동일시하게 나온 점에 대해서는 굉장히 의문이 생기는 실험입니다. SSD와 RAM 그것도 DAX을 이용한 비교하는 것 치고는 굉장히 차이가 적게 나왔기 때문입니다. 생각하던 것의 이하로 나온 결과에 대해서는 의문이 남았습니다.

## **9.11 추가실험** 해당 실험에서 실수가 없음을 확인하였습니다. 그렇다면 실험환경에 문제가 생겨서 차이가 미미하거나 나지 않는 것이 아닐지 궁금하게 되었습니다. 그렇기에 기존 Linux가 설치되어 있는 스토리지에서 실험을 진행해 보았습니다.

|  |  |
| --- | --- |
| Storage | |
| Test SSD | Installed Linux |
| Samsung SSD 860 PRO 256GB | TeamGroup L3 EVO (240GB) |

Test SSD의 경우 기존 테스트하던 환경의 SSD이며  
 Installed Linux의 경우 이번에 추가적으로 실험해 보려고 하는  
 현재 리눅스가 설치된 저장장치입니다.  
그림 30 free 명령어 실행(할당된 메모리가 제외된 총 8GB만 사용 가능)  
실험에 앞서 기존 실험에 영향을 미칠 것 같은 부분이 하나 더 생각이 나서 확인을  
해보았습니다.  
실험은 언제나 DAX로 인하여 할당된 메모리가 제외된 나머지 메모리로 실행이  
되어야 됩니다.  
그런데 만약에 DAX 사용 중일 때만 할당된 메모리가 제외되고  
SSD로 실험할 때는 DAX 마운트 해제로 인한 할당된 메모리가 제외가 되지 않는다면  
실험에 영향이 분명 있을 것이라고 생각이 들어서 확인을 해 보았습니다만  
부팅 이후 아무 작업도 하지 않은 상태에서도 이미 할당된 메모리는 전체 메모리에서  
제외되어서 사용되고 있는 것을 알 수 있었기에 이는 영향을 끼치지 않는다는  
결론이 나왔습니다.  
  
그림 31 main 폴더 추가  
기존에 실험하던 내 문서안에 ssd 그리고 ramdisk 옆에 main 폴더를 추가해주었습니다.  
그림 32 YCSB 실행(main)  
실험은 workloadA 변경없이 그대로 진행했습니다.  
명령어에서는 기존 ssd 혹은 ramdisk로 해 두었던 폴더 위치를 새롭게 만든  
main 폴더로 변경하여 실행하였습니다.

그림 33 load 완료(main)  
해당 저장장치에서도 정상적으로 load가 완료되었습니다.

그림 34 run 실행(main)  
run 실행 또한 정상적으로 실행되었습니다.

## **9.12 추가 결과(WorkloadA)** 읽기 50% 업데이트 50% 추가 실험 결과를 기존의 SSD와 DAX에 비교했을 때에 Main 저장장치를 추가했습니다. 추가 실험 결과 한눈에 보더라도 확실히 차이가 났고 1000배수 인 것을 감안한다면 대략 20000가량의 Throughput의 차이가 나는 것을 확인할 수 있었습니다.

10 참고자료

10.1 사이트

<https://mozi.tistory.com/124>  
  
<https://harryp.tistory.com/662?category=957271>  
  
<https://fio.readthedocs.io/en/latest/fio_doc.html#i-o-depth>  
  
<https://hiseon.me/linux/linux-ramdisk>  
  
<https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/configuring-persistent-memory-for-file-system-direct-access-dax>  
  
<https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/ch-persistent-memory-nvdimms#sect-Configuring-Persistent-Memory-with-ndctl>  
  
<https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/configuring-persistent-memory-for-use-in-device-dax-mode>  
  
<https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/ch-ext4#ext4creating>  
  
<https://m.blog.naver.com/PostView.nhn?blogId=sysganda&logNo=30183591170&proxyReferer=https%3A%2F%2Fwww.google.com%2F>  
  
<http://blog.naver.com/PostView.nhn?blogId=sysganda&logNo=30183365906&parentCategoryNo=&categoryNo=41&viewDate=&isShowPopularPosts=false&from=postView>  
  
<https://sekwonlee.blogspot.com/2014/07/filebench.html>  
<https://pmem.io/2016/02/22/pm-emulation.html>  
  
<https://twinw.tistory.com/55>  
<https://hyunyoung2.github.io/2016/03/24/rocksDB_Install2/>  
  
<https://blog.naver.com/PostView.nhn?blogId=hj_veronica&logNo=220696840819&proxyReferer=https%3A%2F%2Fwww.google.com%2F>