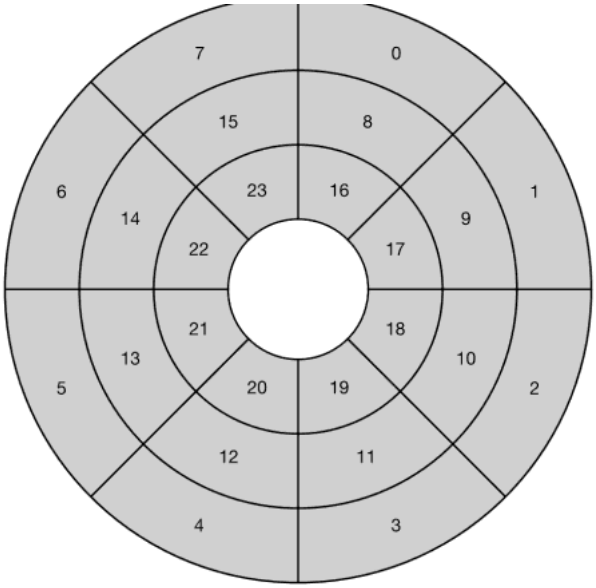
< HDD, SDD의 데이터와 성능비교 >

2018213354
컴퓨터공학과
오 은 비
2021.10.18

저장 장치의 대표격인 HDD의 데이터 읽기 쓰기의 동작방식

HDD는 데이터를 자기 정보로 변환하여 platter라고 불리는 자기 장치에 기록한다. (이 때 데이터는 byte단위가 아닌 sector단위이다.) 밑의 그림과 같이 섹터는 원의 중심부터 바깥방향으로 분할되어 있으며, 각각에 주소번호가 있다.

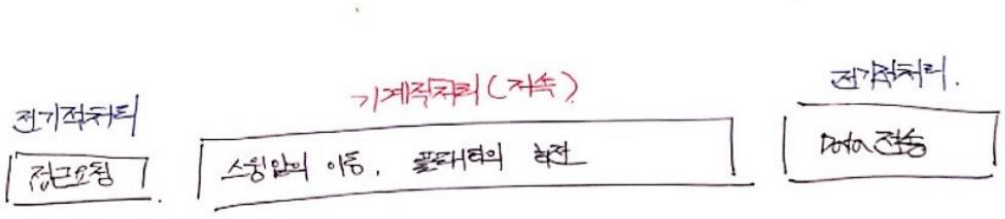


플래터의 각 섹터는 자기 헤드라는 부품에 의해 읽고 쓰여진다. 자기 헤드는 스윙 암이라는 부품에 달려있고 그것이 움직임으로써 자기 헤드를 플래터의 원 반경 방향으로 이동시킨다. 플래터를 회전시킴으로써 자기 헤드를 읽고 싶은 대상 섹터의 바로 위에 오도록 한다. 따라서 HDD로부터의 데이터 전송흐름은

1. 데이터의 읽고 쓰기에 필요한 정보(섹터 번호, 섹터의 개수, 섹터의 종류 등)를 HDD에 전달한다.
2. 스윙 암을 이동시키거나 플래터를 회전시켜서 접근하려고하는 섹터 위에 자기 헤드를 위치시킨다.

3. 데이터를 읽고 쓴다
4. 읽을 경우에는 HDD의 읽기 처리가 완료 된다.

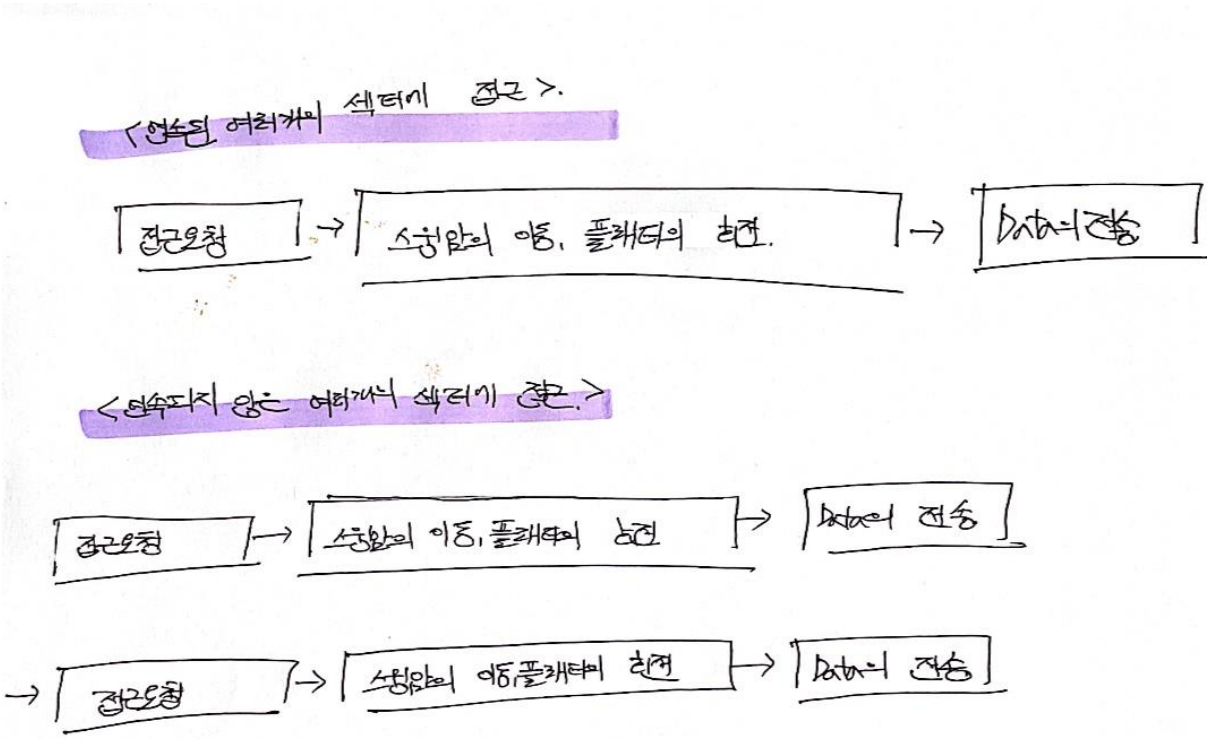
위에서 설명한 전송흐름중 1번과 4번은 고속의 전기적 처리인 반면, 스윙 암의 동작과 플래터의 회전은 기계적 처리라서 훨씬 느리다. 따라서 HDD에 접근하는 레이턴시는 하드웨어의 처리 속도에 영향을 받는다. 접근 처리에 드는 소요시간을 그림으로 나타내면



위의 그림에서 보면 알 수 있듯이 대부분의 속도가 기계적처리(저속)으로 진행됨을 알 수 있다.

HDD의 성능 특성

HDD는 연속하는 여러 개의 섹터 데이터를 한 번의 접근 요청으로 읽을 수 있다. 스윙 암을 동작해서 자기 헤드의 위치를 동심원의 위치에 맞추기만 하면 플래터를 회전시키는 것만으로 여러 데이터를 읽을 수 있기 때문이다. 그러나 한 번에 읽을 수 있는 양은 HDD마다 제한이 있음에 유의해야한다. 또 여러 개의 섹터가 연속적으로 있더라도 각각의 섹터를 여러 번 나눠서 접근한다면 HDD에 여러 번 접근 요청을 해야하기때문에 불필요한 시간이 소요된다. 이 때 소요시간을 시간순으로 나타낸다면



HDD는 이러한 성능 특성 때문에 파일시스템은 각 파일의 데이터를 가능한 연속된 영역에 배치되도록 해야한다. 또한 연속된 영역에 접근할 때에는 한 번에 이루어져야하고 파일은 되도록 큰 사이즈로 sequential하게 접근해야한다.

HDD의 테스트

HDD의 성능 측정을 실험을 통해 확인해본다. 움직이고 있는 데이터를 측정해야하기 때문에 블록 장치의 데이터를 직접 읽고, 내 컴퓨터의 데이터가 파괴되지 않도록 사용하지 않는 파티션을 준비해 이 테스트에 사용되도록 하였다.

이 테스트에서는 I/O size에 따른 성능 변화와 시퀀셜 접근과 랜덤 접근의 차이를 측정할 것이다. 이를 위해 지정된 파티션의 처음부터 1Gbyte까지의 영역에 합계 64Mbyte의 I/O를 요청한 시퀀셜 혹은 랜덤접근, 1회당 I/O size를 지정할 수 있게 했다.

<소스코드>

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <linux/fs.h>

#define PART_SIZE (1024*1024*1024)
#define ACCESS_SIZE (64*1024*1024)

static char *progname;

Int main(int argc, char *argv[])
{
        progname = argv[0];
        if(argc !=6){
```

```
fprintf(stderr, “usage: %s <filename> <kernel’s help> <r/w>
        <access pattern> <block size[KB]>\n”, progname);
        exit(EXIT_FAILURE);

}

char *filename = argv[1];

bool help;
If(!strcmp(argv[2], “on”)) {
        help = true;
}

else if (!strcmp(argv[2], “off”)) {
        help = false;
}

else{
        fprintf(stderr, “kernel’s help should be ‘on’ or ‘off’ : %s\n”,
        argv[2]);
        exit(EXIT_FAILURE);
}


int write_flag;
if(!strcmp(argv[3],”r”)) {
        write_flag = false;
}

else if(!strcmp(argv[3],”w”)) {
        write_flag = true;
}

else{
        fprintf(stderr, “r/w should be ‘r’ or ‘w’ : %s \n”,
        argv[3]);
        exit(EXIT_FAILURE);
}
```

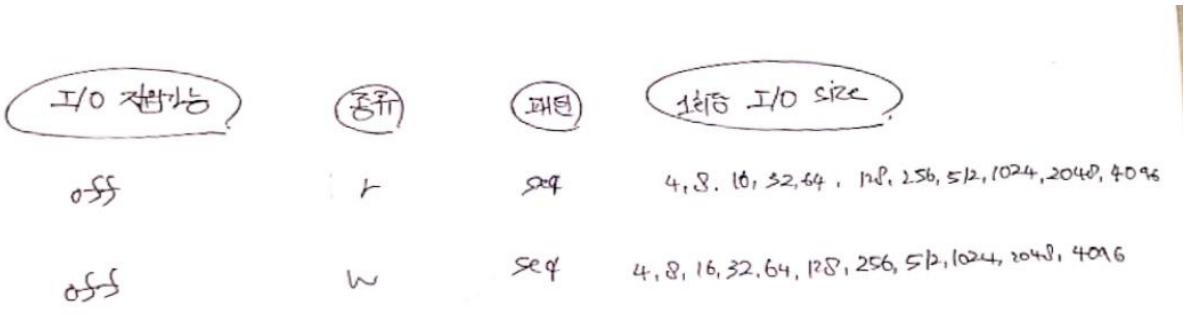
```
bool random;
if(!strcmp(argv[4],”seq”)) {
        random = false;
}

else if(!strcmp(argv[4],”rand”)) {
        random = true;
}

else{
        fprintf(stderr, “access pattern should be ‘seq’ or ‘rand’ : %s \n”,
        argv[4]);
        exit(EXIT_FAILURE);
}

int part_size = PART_SIZE;
int access_size = ACCESS_SIZE;

int block_size = atoi(argv[5]) * 1024;
if( block_size==0){
        fprintf(stderr, “block size should be > 0: %s\n”,
        argv[5]);
        exit(EXIT_FAILURE);
}

if(access_size % block_size !=0) {
        fprintf(stderr, “access size(%d) should be multiple of block
        size : %s\n”, access_size, argv[5]);
        exit(EXIT_FAILURE);
}

int maxcount = part_size / block_size;
int count = access_size / block_size;

int *offset = malloc(maxcount * sizeof(int));
if (offset == NULL)
        err(EXIT_FAILURE, “malloc() failed”);

int flag =0_RDWR | 0_EXCL;
if(!help)
```

```
        flag |= 0_DIRECT;

int fd;
fd = open(filename, flag);
if(fd == -1)
        err(EXIT_FAILURE, “open() failed”);

int i;
for(i = 0; i < maxcount; i++) {
        offset[i] = i;
}

if(random) {
        for(i = 0; i < maxcount; i++) {
                int j = rand() % maxcount;
                int tmp = offset[i];
                offset[i] = offset[j];
                offset[j] = tmp;
          }
}

int sector_size;
if (ioctl(fd, BLKSSZGET, &sector_size) == -1)
        err(EXIT_FAILURE, "ioctl() failed");

char *buf;
int e;
e = posix_memalign((void **)& buf, sector_size, block_size);
if(e) {
        errno = e;
        err(EXIT_FAILURE, "posix_memalign() failed");
}

for( i = 0; i < count; i++){
        ssize_t ret;
        if(lseek(fd, offset[i] * block_size, SEEK_SET) == -1)
                err(EXIT_FAILURE, "lseek() failed");
        if(write_flag){

                ret = write(fd, buf, block_size);
```

```
                        if(ret == -1)
                                err(EXIT_FAILURE, "write() failed");
        }

        else {
                ret = read(fd, buf, block_size);
                if(ret == -1)
                        err(EXIT_FAILURE, "write() failed");
         }
        else {
                ret = read(fd, buf, block_size);
                if( ret == -1)
                        err(EXIT_FAILURE, "read() failed);
        }
}

if (fdatasync(fd) == -1)
        err(EXIT_FAILURE, "fdatasync() failed");
if(close(fd) == -1)
        err(EXIT_FAILURE, "close() failed");

exit(EXIT_SUCCESS);

}
```

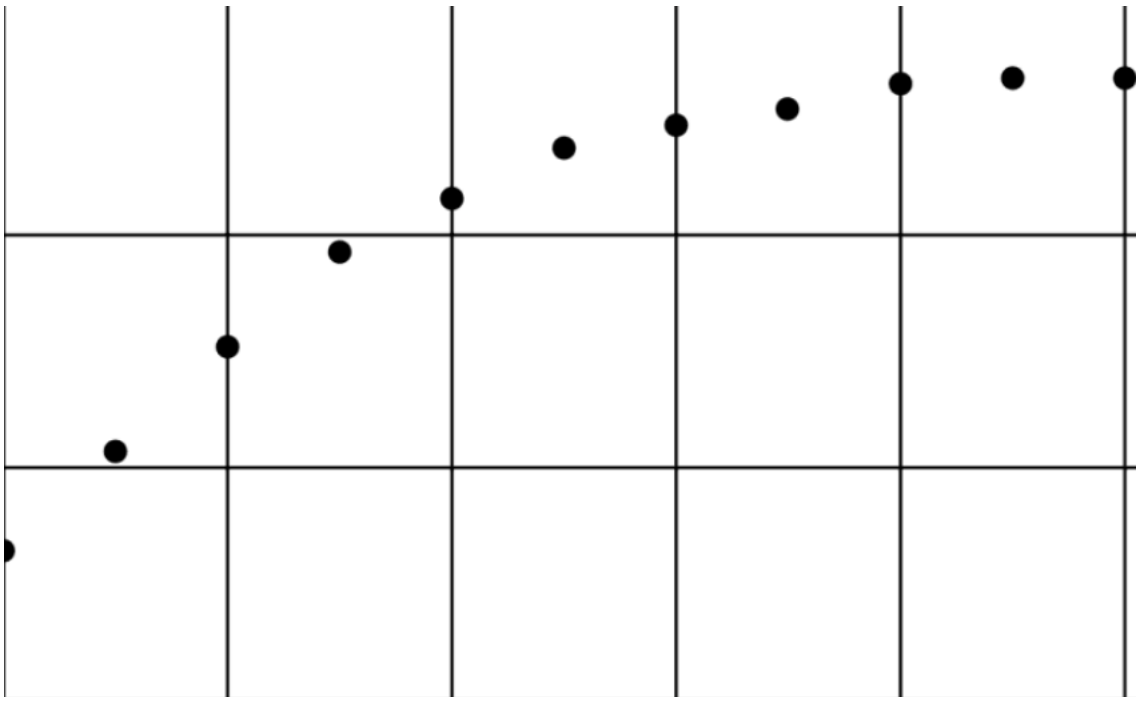
시퀀셜 접근(I/O지원기능 Off일 경우)

다음 파라미터로 데이터를 얻는다고 가정한다.

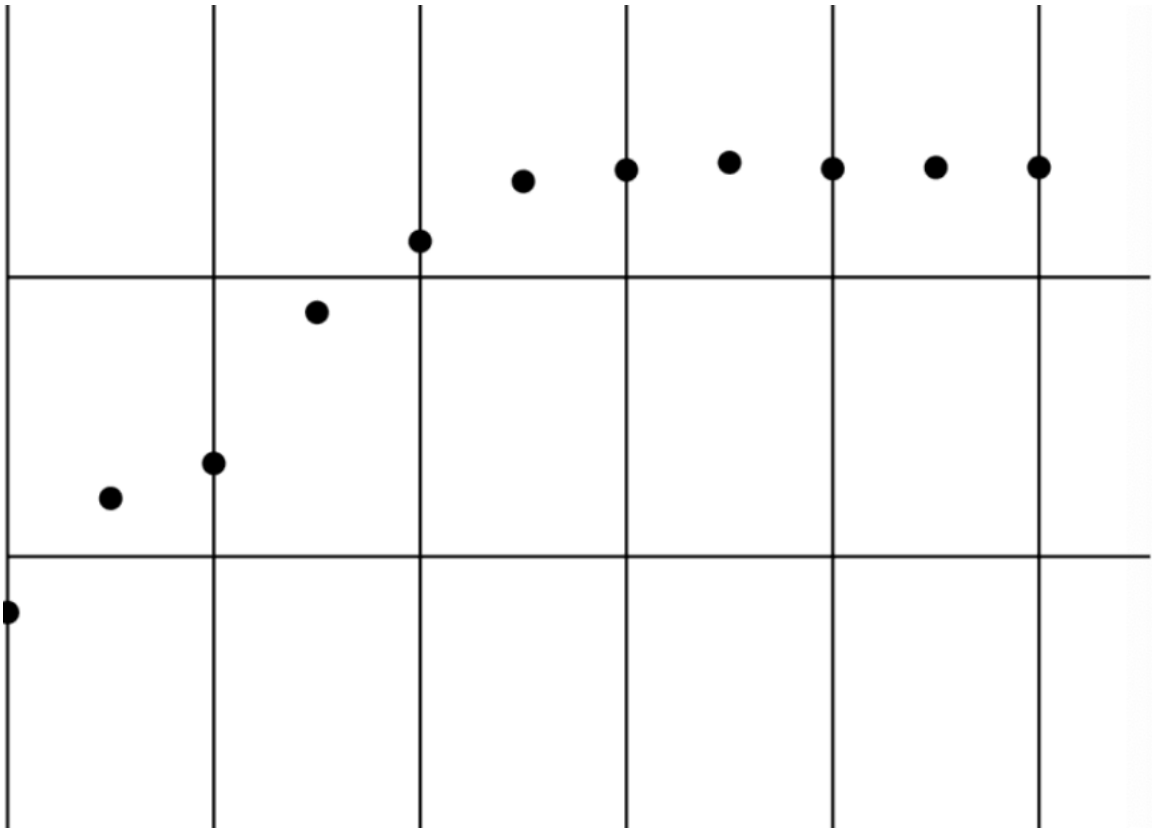


읽기와 쓰기 각각에 대한 그래프를 그래보면 (세로축이 throughput, 가로축은 I/O size)

지원 기능을 껐을 때의 HDD의 시퀀셜 읽기 성능은 아래 그림과 같고



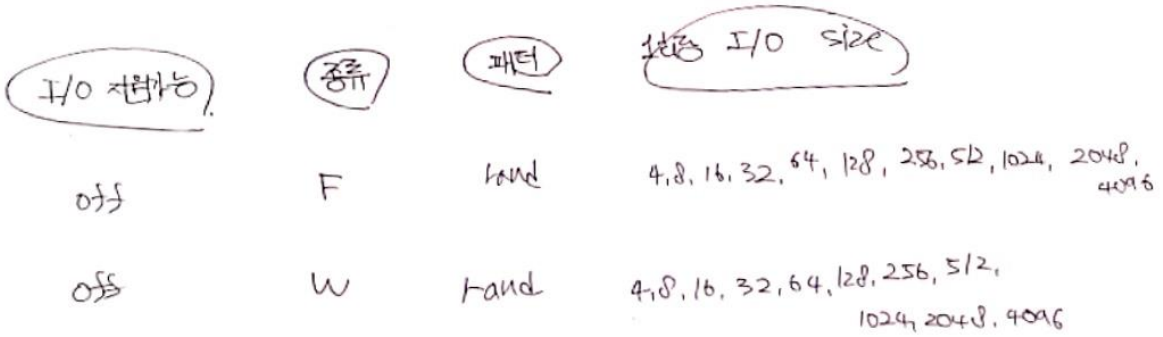
지원 기능 껐을 때의 HDD의 시퀀셜 쓰기 성능은 아래 그림과 같다



표를 보면 읽기와 쓰기 모두 1회당 I/O size가 커질수록 throughput성능도 향상됨을 알 수 있다. 그러나 I/O size가 1Mbyte로 되었을 때부터 성능이 더 이상 올라가지 않음을 볼 수 있다. 이것은 이 HDD가 한 번에 접근할 수 있는 데이터량의 한계이고 이때의 스루풋 성능이 이 HDD의 최대성능이기 때문이다.

랜덤접근

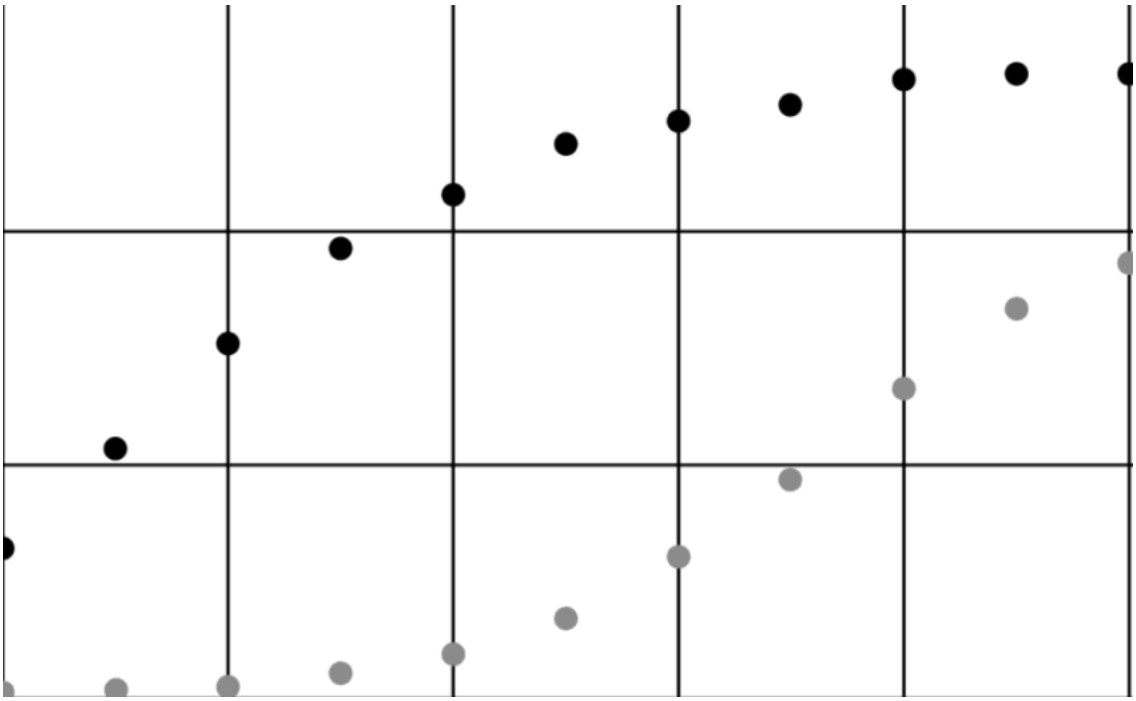
다음 파라미터로 데이터를 얻는다고 가정한다.



결과를 시퀀셜 접근과 비교한다.

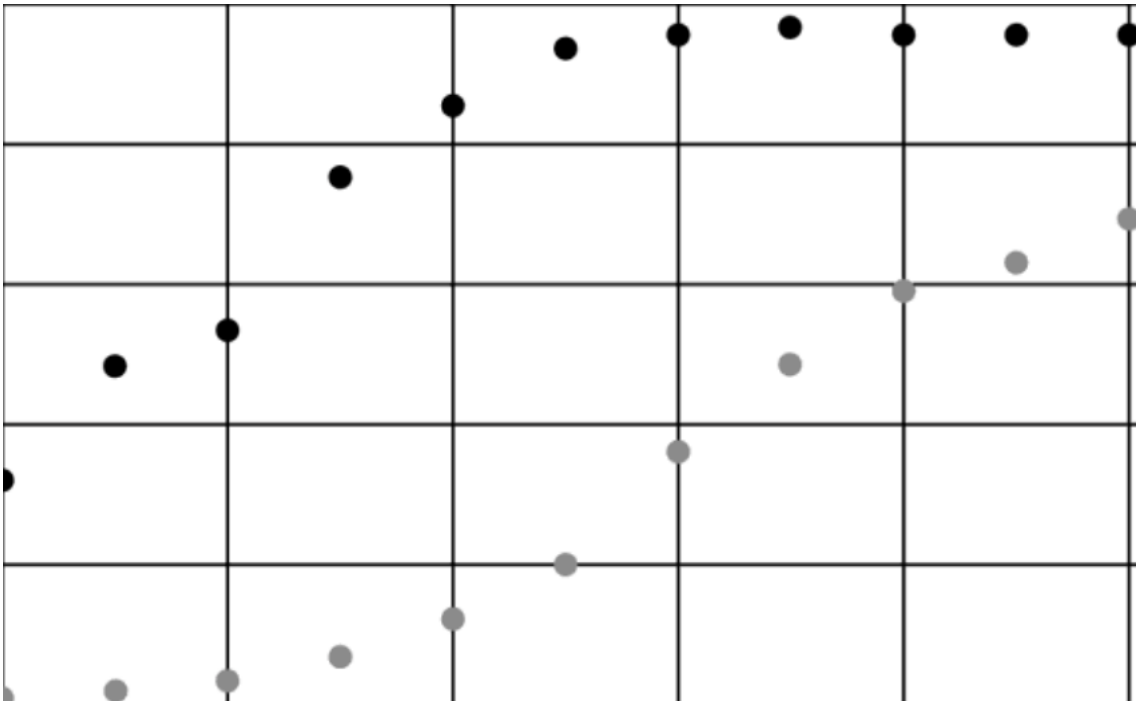
HDD의 읽기 성능(지원 기능 off)

(검은점은 시퀀셜, 회색점은 랜덤)



HDD의 쓰기 성능(지원기능 off)

(검은점은 시퀀셜, 회색점은 랜덤)



그래프를 보면 랜덤접근의 성능이 시퀀셜 성능보다 뛰어남을 알 수 있다. 특히 I/O 사이즈가 작을 때 성능의 차이가 심해지는 것 같다.

시퀀셜 접근(I/O지원기능 On일 경우)

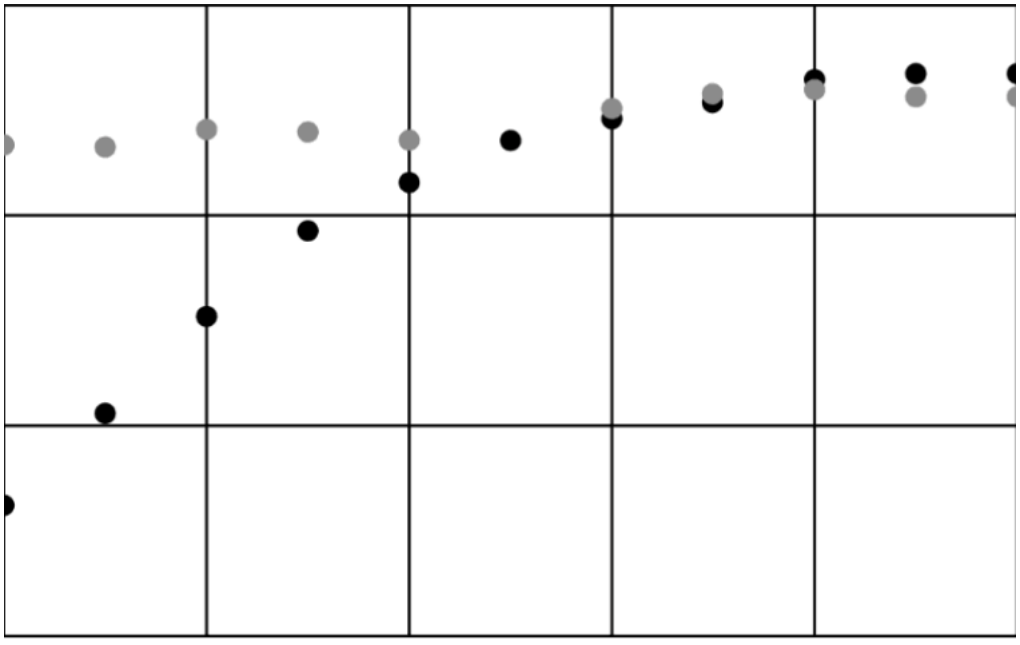
다음 파라미터로 데이터를 얻는다고 가정한다.

I/O 지원기능	종류	패턴	전송 I/O size
On	r	Seq	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096
On	w	Seq	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096

각각의 결과를 I/O지원기능이 없었던 때와 비교해보면

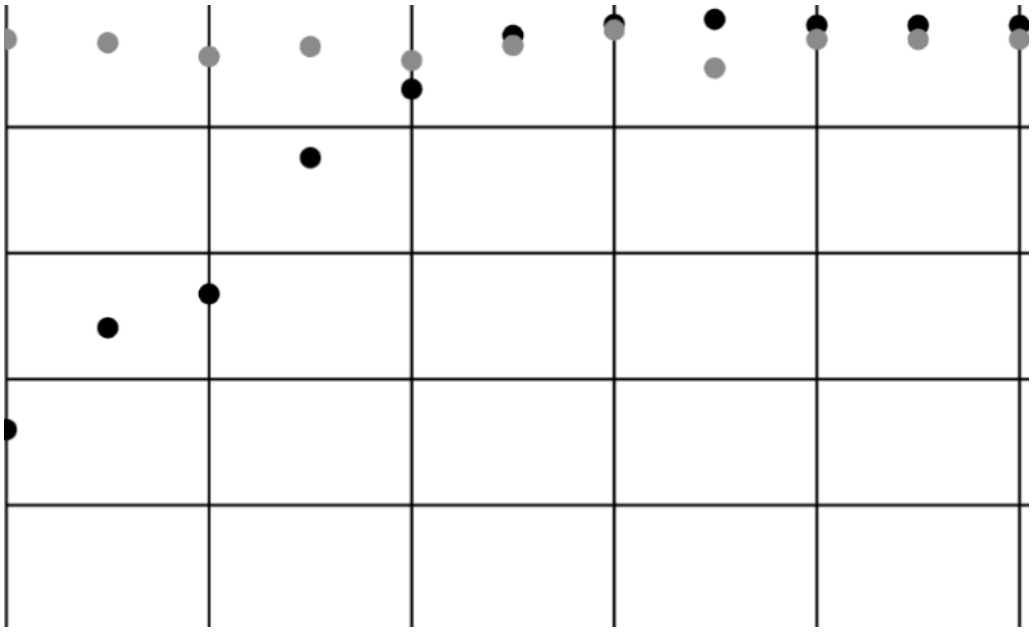
시퀀셜 읽기 성능 비교

(검은 점은 지원기능off, 회색점은 지원기능on)



시퀀셜 쓰기 성능 비교

(검은 점은 지원기능off, 회색점은 지원기능on)



읽기와 쓰기 모두 I/O 사이즈가 작은 시점부터 throughput성능이 HDD의 한계에 거의 다다른 것을 알 수 있다. 이 것이 미리 읽기의 효과이다. 처음의 데이터에 접근한 시점에 다음의 연속된 데이터를 미리 읽기 때문에 이렇게 나타난다. (따라서 자동적으로 뒤에 연속해서 읽는 영역에 대해서는 이미 데이터가 메모리에 있기 때문에 소요 시간이 짧아지게 된다.)

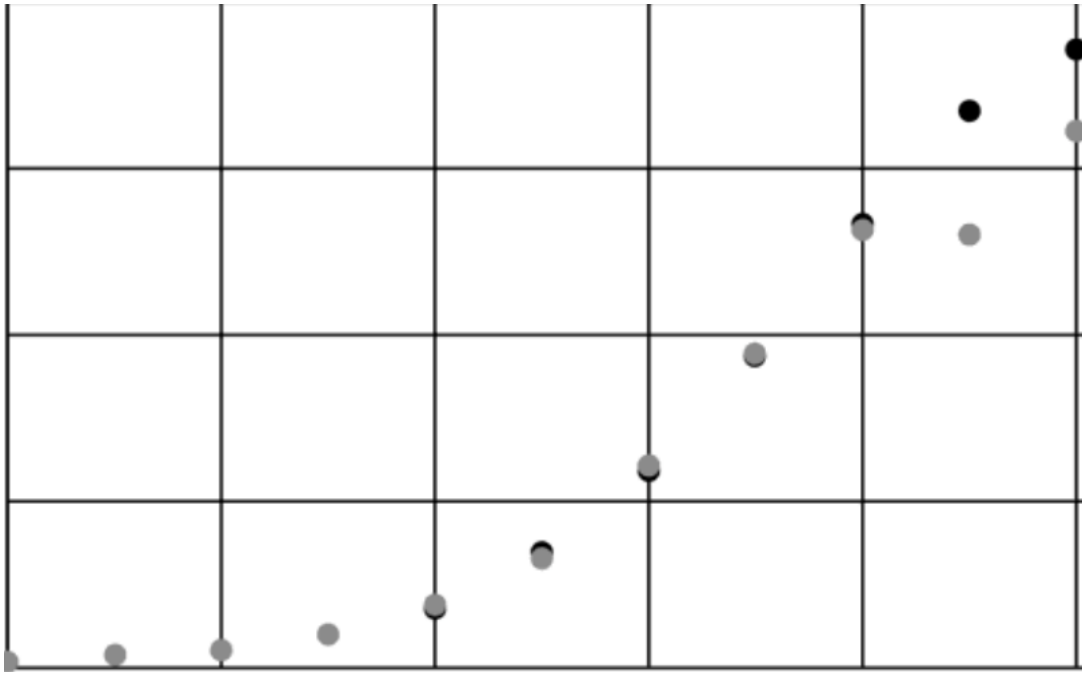
랜덤접근

다음 파라미터로 데이터를 얻는다고 가정한다.

I/O 지원기능	종류	패턴	1타입 I/O size
On	r	rand	4,8,16,32,64,128,256,512,1024,2048,4096.
On	w	rand	4,8,16,32,64,128,256,512,1024,2048,4096

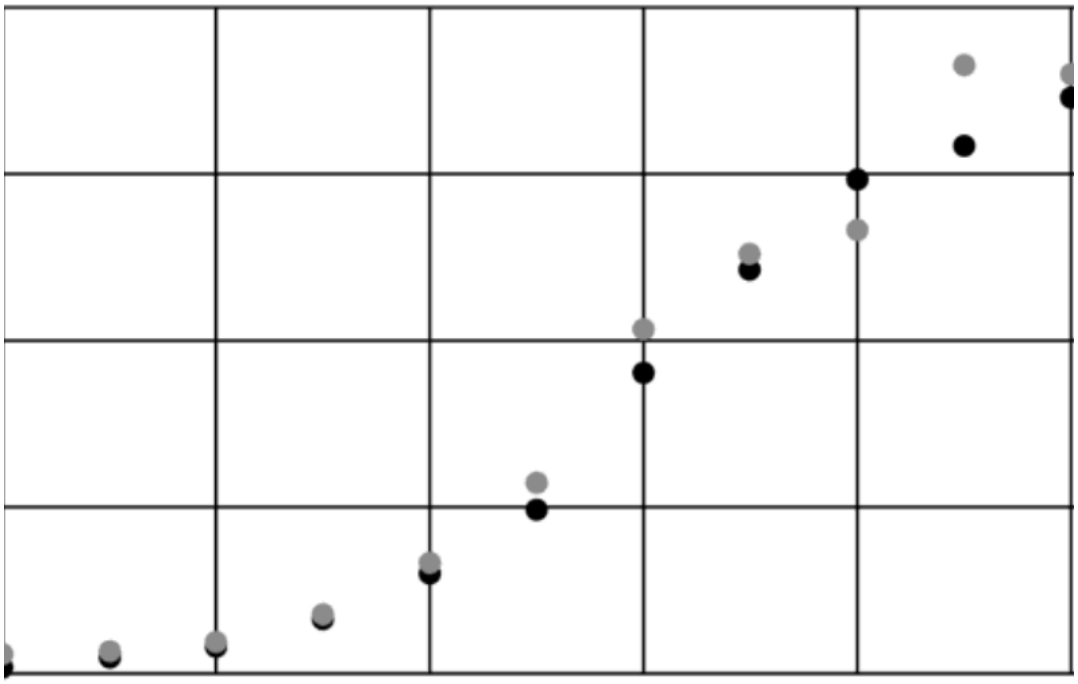
랜덤접근 읽기성능 비교(지원기능on)

(검은 점은 지원기능off, 회색점은 지원기능on)

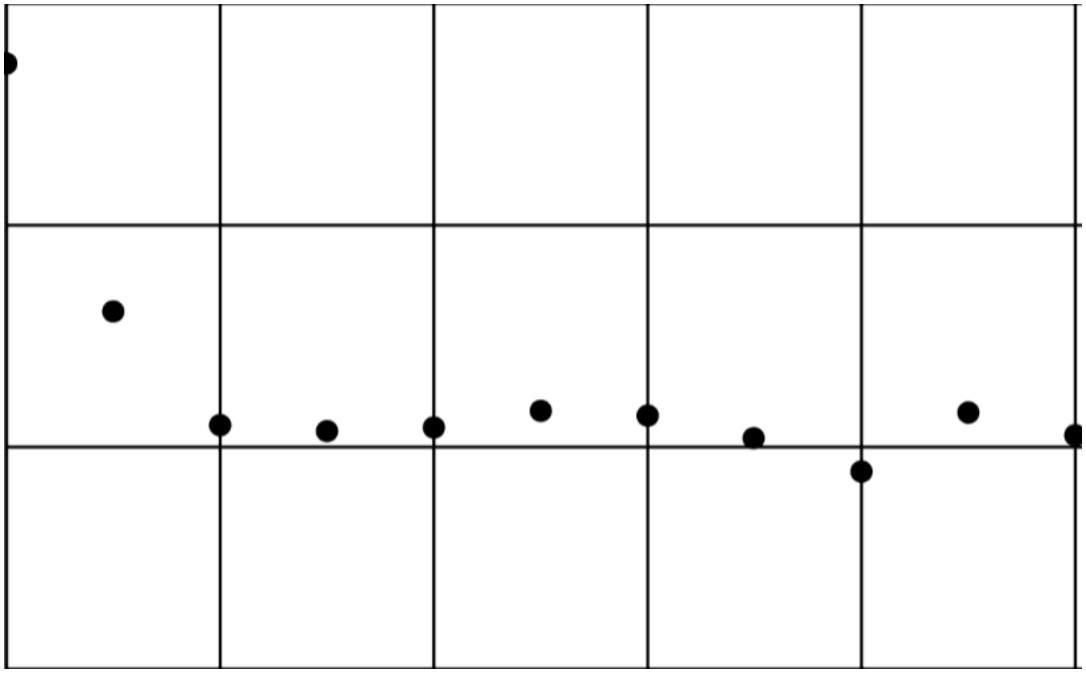


랜덤접근 쓰기성능 비교

(검은 점은 지원기능off, 회색점은 지원기능on)



두 경우 모두 I/O size가 커질수록 throughput성능이 시퀀셜에 가까워지다가 결국 같아짐을 알 수 있다. 읽기의 랜덤 접근은 거의 바뀌지 않고 있는데 그 이유는 I/O스케줄러가 동작하지 않는 상태로 미리 읽기를 하더라도 시퀀셜 접근이 되지 않으므로 결국 미리 읽기 한 데이터가 버려지기 때문이다. 쓰기는 이 표에서는 알아보기 힘들지만 I/O 사이즈가 작은 경우 I/O의 스케줄러가 보인다. 이를 육안으로 확인해보기 위해 세로축을 I/O 지원 기능을 켠 경우의 throughput 성능/ 끈 경우의 throughput성능으로 나타내보면



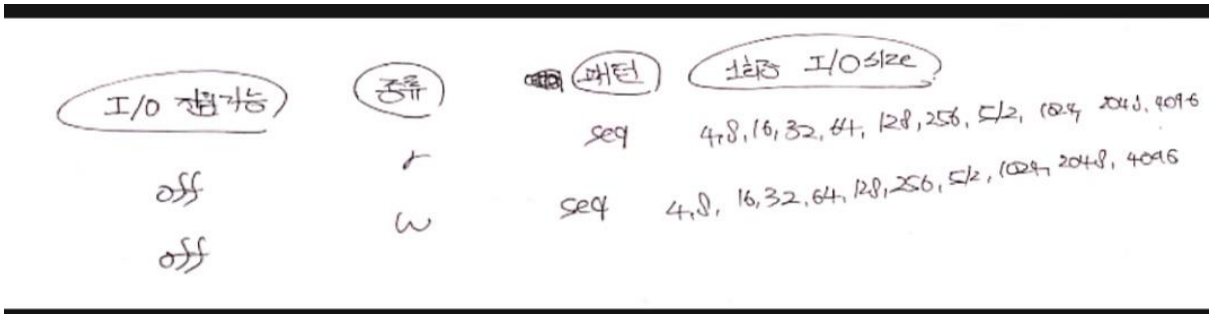
이렇게 나타나게 된다.

SSD의 동작 방식

SSD와 HDD의 가장 큰 차이점은 SSD는 데이터에 접근하는 것이 전기적 동작만으로 이루어진다는 점이다. 따라서 랜덤 접근 성능도 SSD가 HDD보다 빠르게 되는 것이다.

SSD 시퀀셜 읽기 성능(지원기능 off)

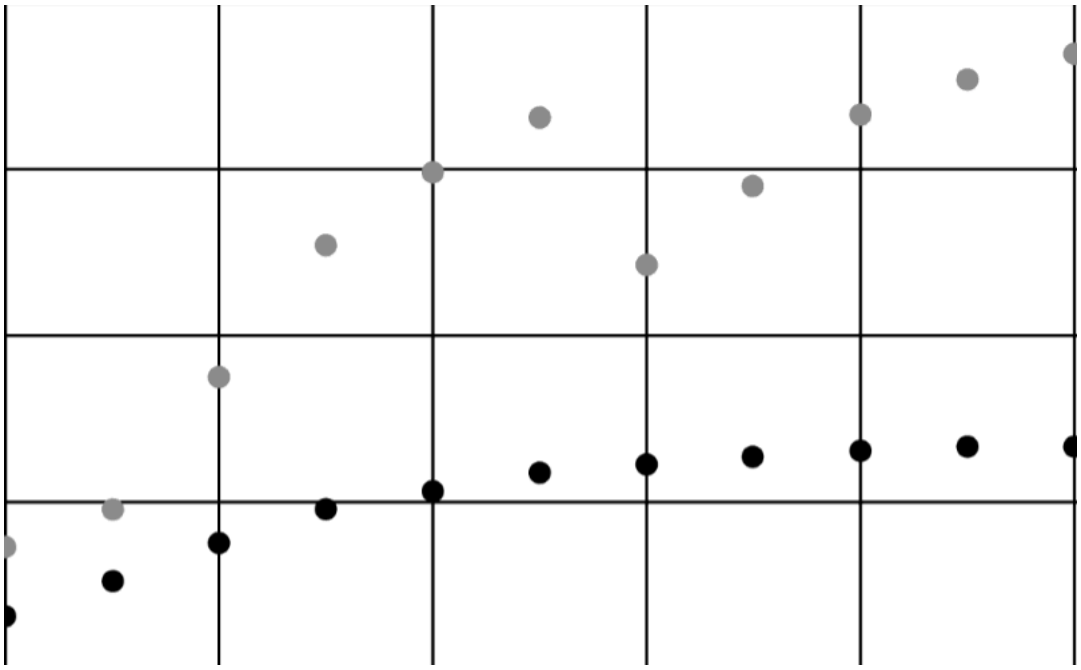
다음 파라미터로 데이터를 얻는다고 가정한다.



테스트 결과를 HDD 데이터와 비교하면

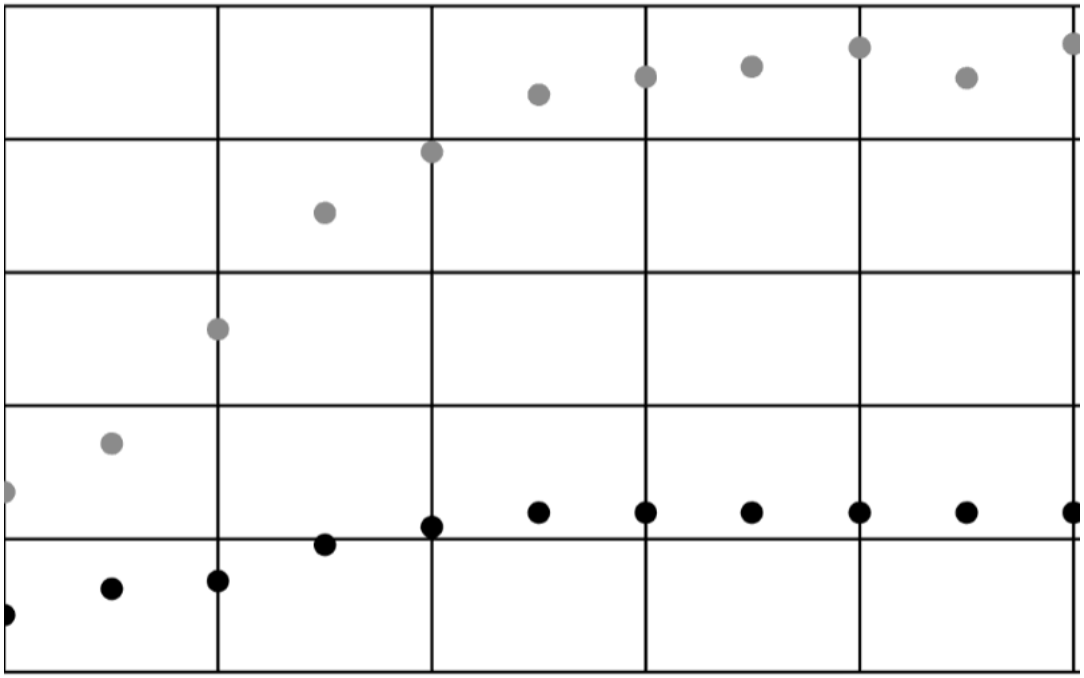
HDD와 SDD의 시퀀셜 읽기성능(지원기능 off)

(검은점 = HDD, 회색점 = SSD)



HDD와 SDD의 시퀀셜 쓰기성능(지원기능 off)

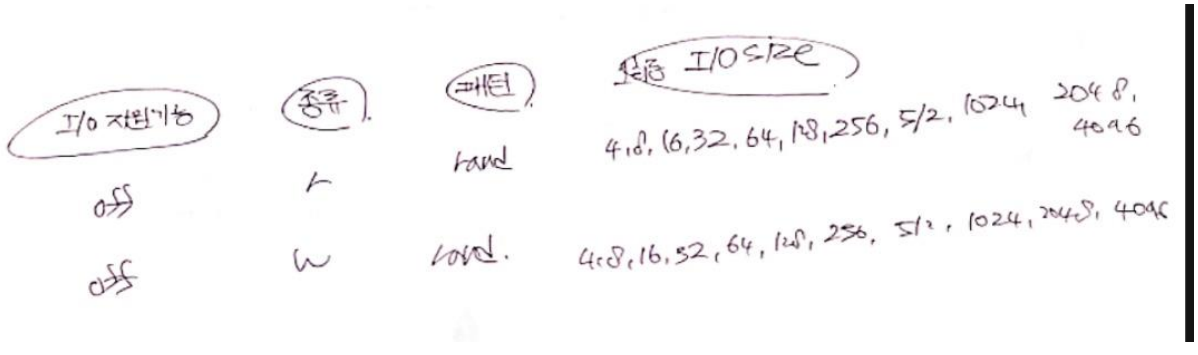
(검은점 = HDD, 회색점 = SSD)



읽기, 쓰기 둘 다 HDD보다 SSD가 빠름을 알 수 있다.

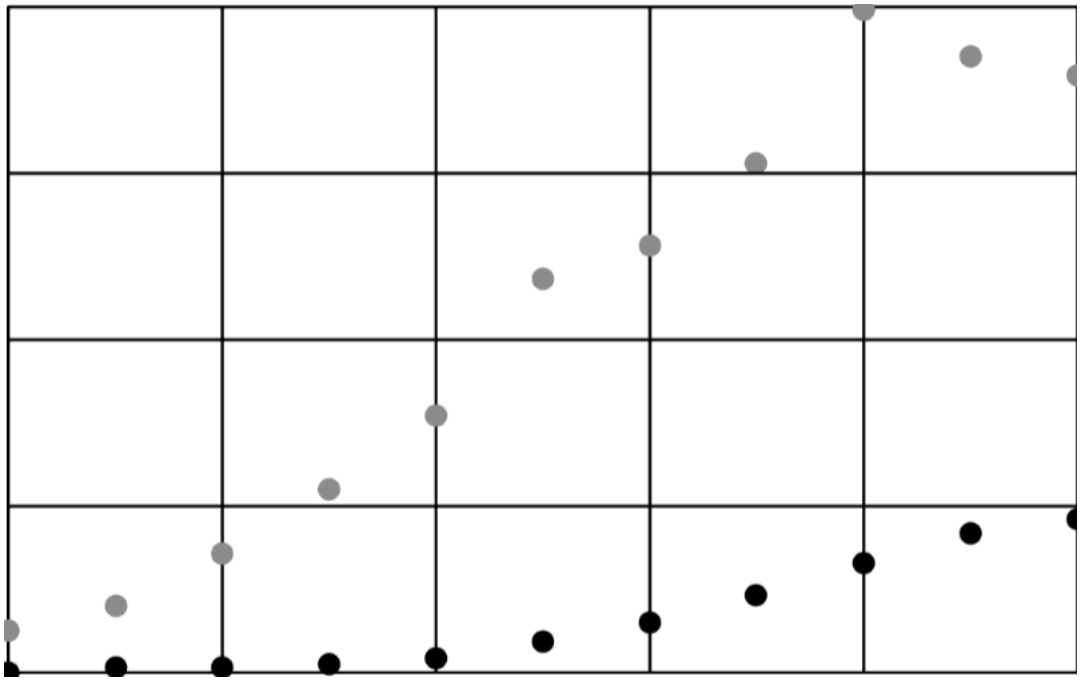
랜덤접근 쓰기성능 비교

다음 파라미터로 데이터를 얻는다고 가정한다.



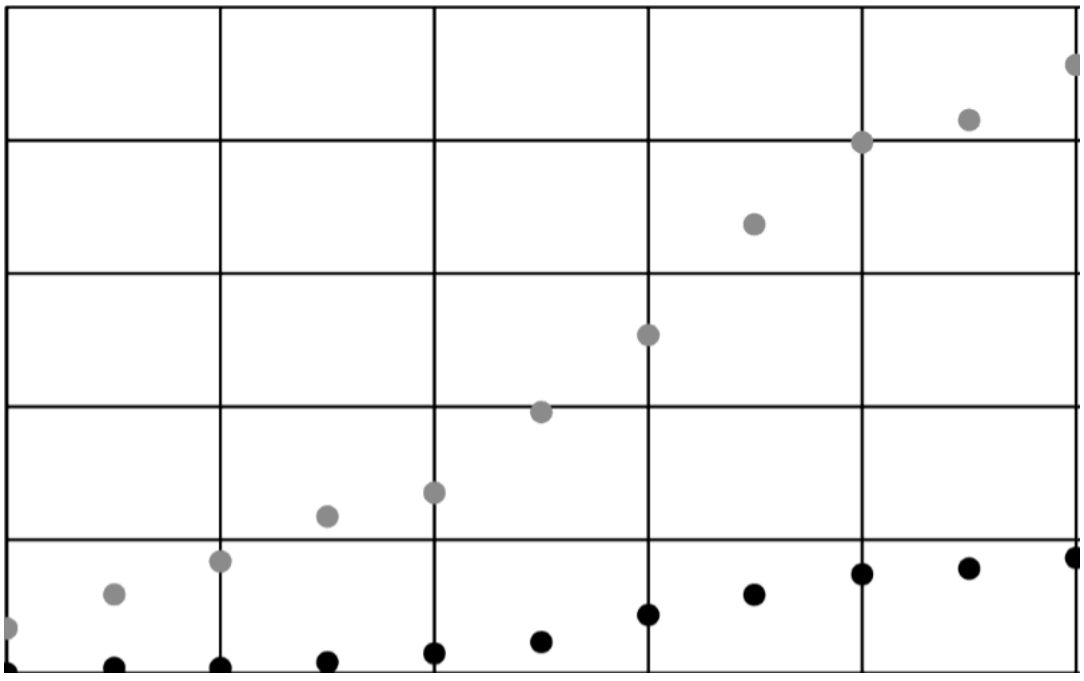
HDD와 SSD의 랜덤 읽기 성능 비교 (지원기능 off)

(검은점 = HDD, 회색점 = SSD)



HDD와 SSD의 랜덤 쓰기 성능 비교 (지원기능 off)

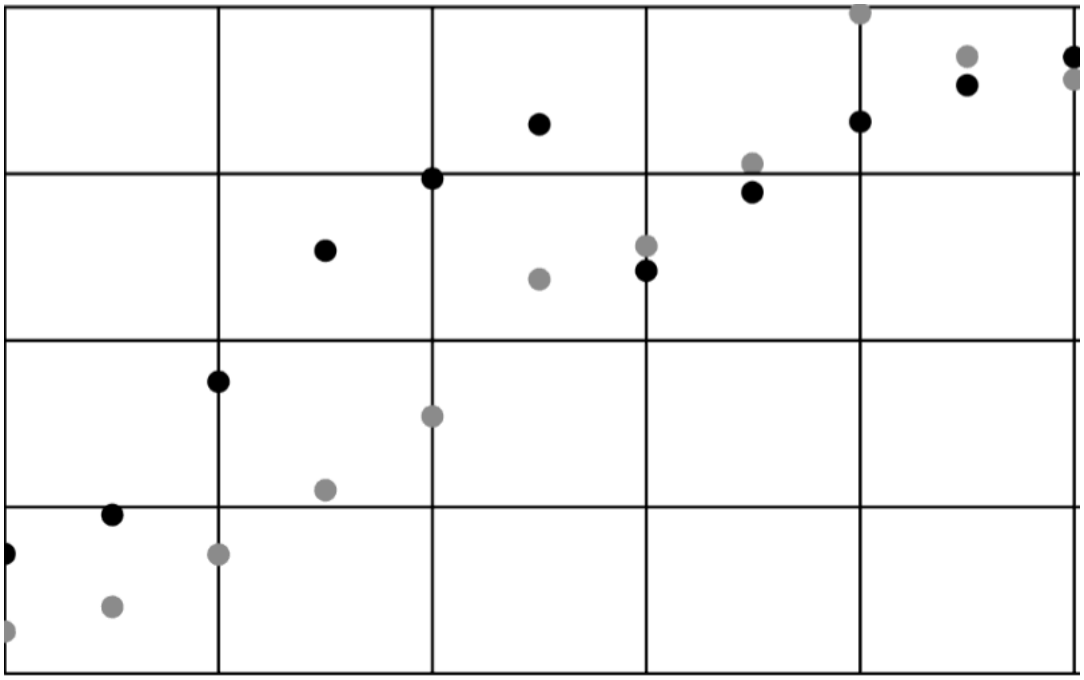
(검은점 = HDD, 회색점 = SSD)



읽기와 쓰기 모두 I/O size가 커질수록 throughput 성능이 높아지는 건 HDD와 같지만 스루풋 성능 차이에서 큰 차이를 보임을 알 수 있다.

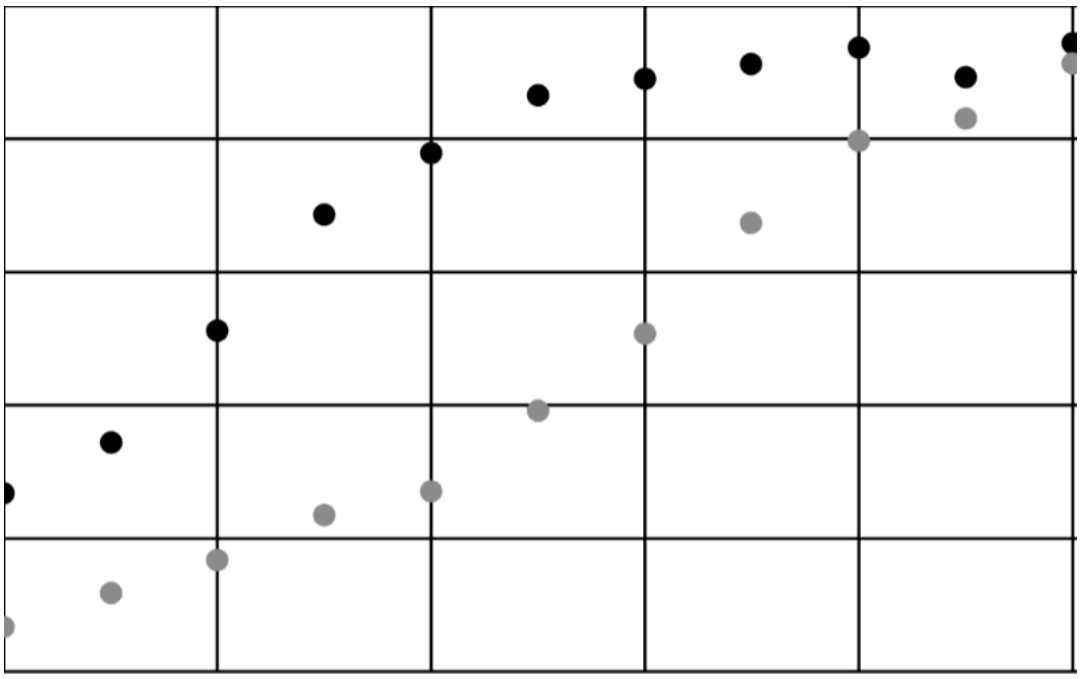
SSD의 읽기 성능 (지원기능off)

(검은점 = 시퀀셜, 회색점 = 랜덤)



SSD의 쓰기 성능 (지원기능off)

(검은점 = 시퀀셜, 회색점 = 랜덤)



게다가 이 두 그래프를 보면 SSD는 시퀀셜접근일때보다 랜덤접근일때 throughput성능이 더 뛰어나짐을 알 수 있다.

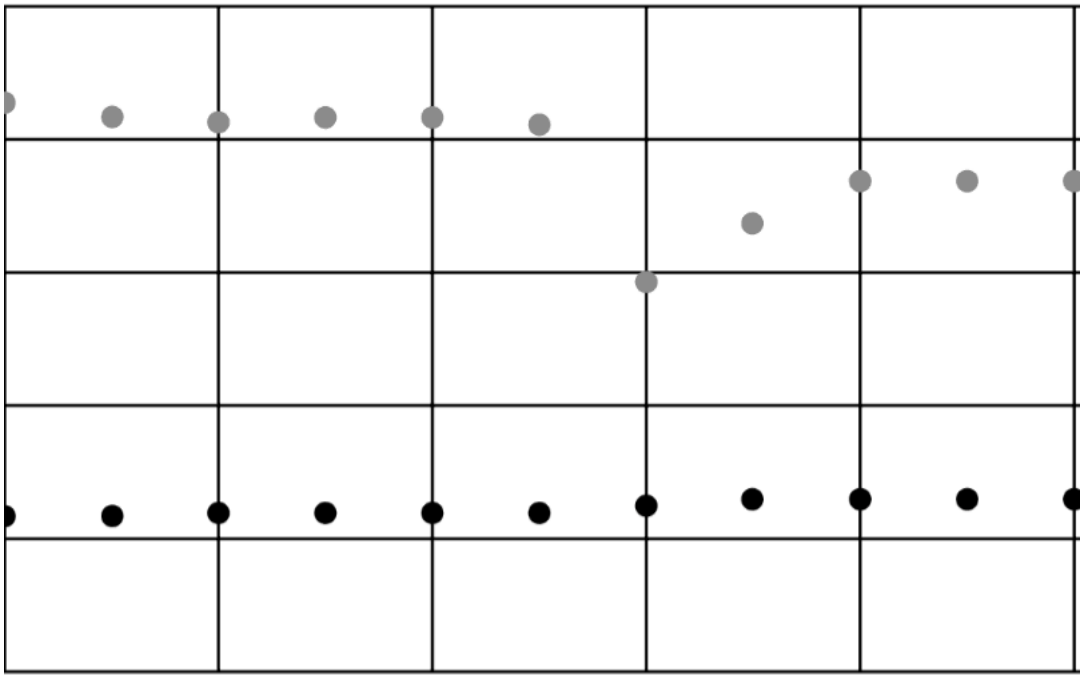
SSD와 HDD의 시퀀셜 성능비교 (지원기능on)

다음 파라미터로 데이터를 얻는다고 가정한다.

I/O 지원기능	종류	패턴	1회당 I/O size
on	r	seq	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096
on	w	seq	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096

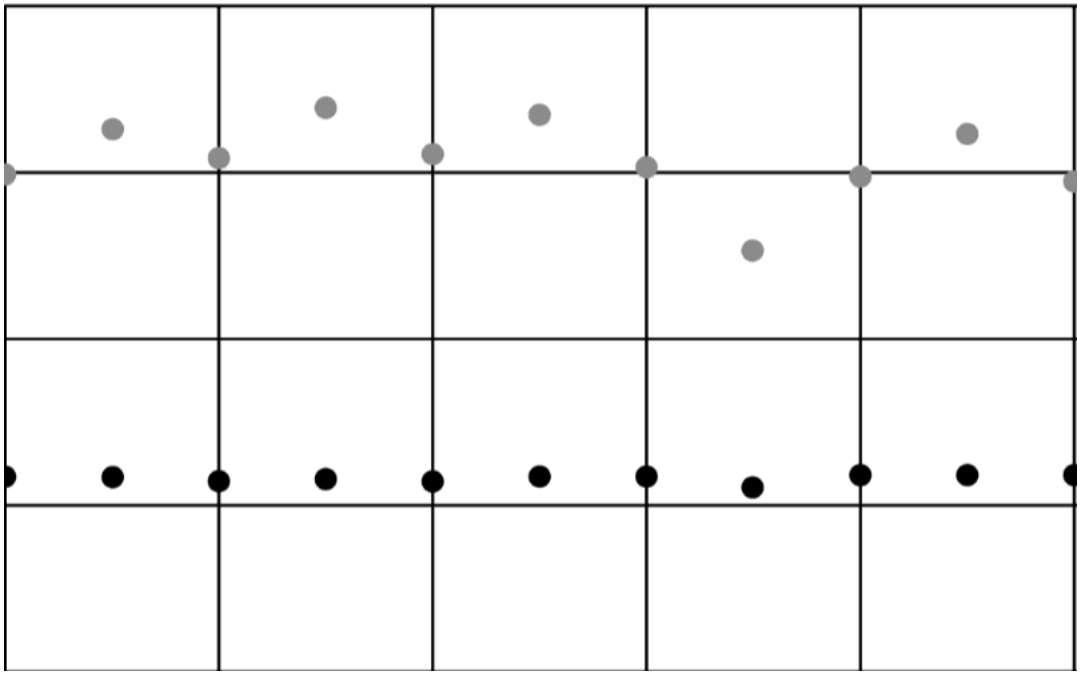
SSD와 HDD의 시퀀셜 읽기성능비교 (지원기능on)

(검은점은 HDD, 회색점은 SSD)



SSD와 HDD의 시퀀셜 쓰기성능비교 (지원기능on)

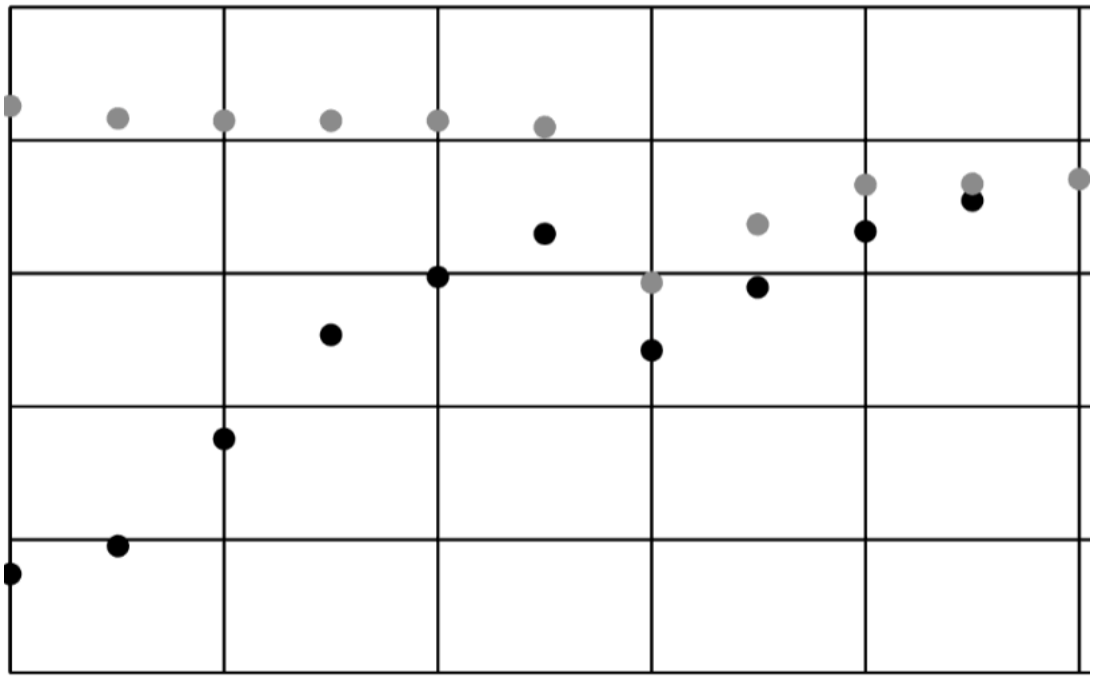
(검은점은 HDD, 회색점은 SSD)



SSD는 HDD와 마찬가지로 작은 I/O size이거나 큰 사이즈이거나 비슷하게 throughput성능이 한계에 접근하고 있음을 알 수 있다. 읽기 성능이 좋은 건 HDD에서 그랬던 것 처럼 미리 읽기의 효과로 보인다. 쓰기 성능에 대해서도 HDD에서 그랬던 것 처럼 I/O 스케줄러의 병합처리효과이다. I/O지원이 있는경우와 없는 경우를 비교해보면

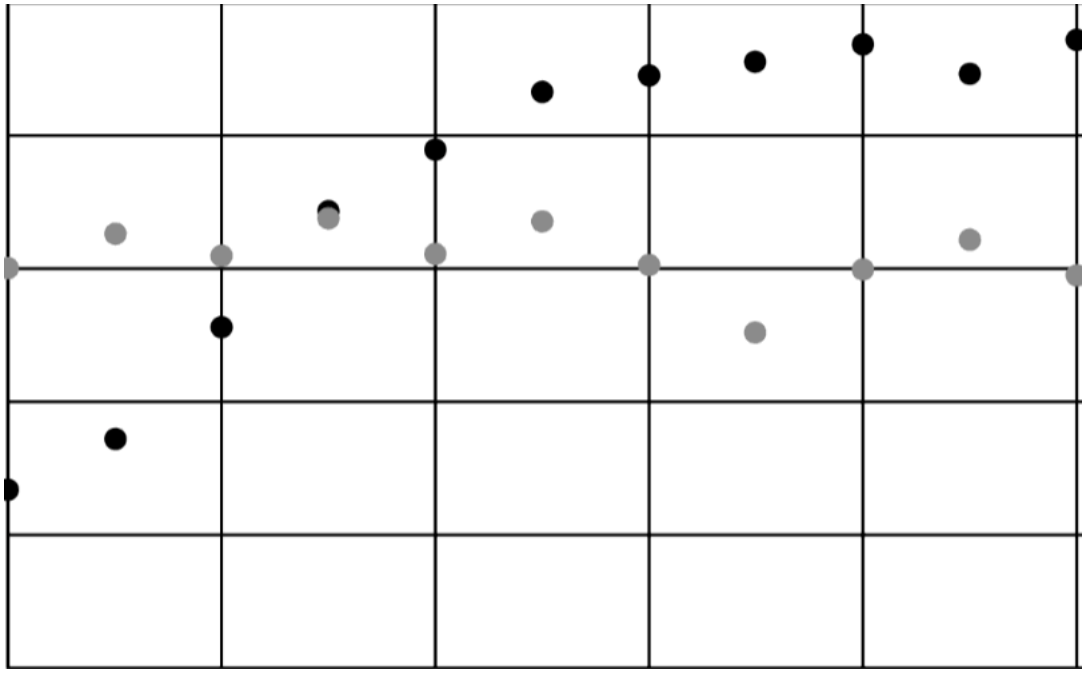
SSD 시퀀셜 읽기 성능

(검은점은 지원기능off, 회색점은 지원기능 on)



SSD 시퀀셜 쓰기 성능

(검은점은 지원기능off, 회색점은 지원기능 on)



쓰기에 대해서 커다란 I/O size의 경우에 오히려 throughput 성능이 낮아진다는 것을 알 수 있다. 그 이유는 I/O 스케줄러의 처리를 하기 위해 여러개의 I/O 요청을 모아두었다가 처리하는 오버헤드가 SSD의 경우 부하되기 때문이다. HDD의 경우에는 기계적 처리의 소요시간이 이 오버헤드보다 크기 때문에 보이지 않았던 것이다.

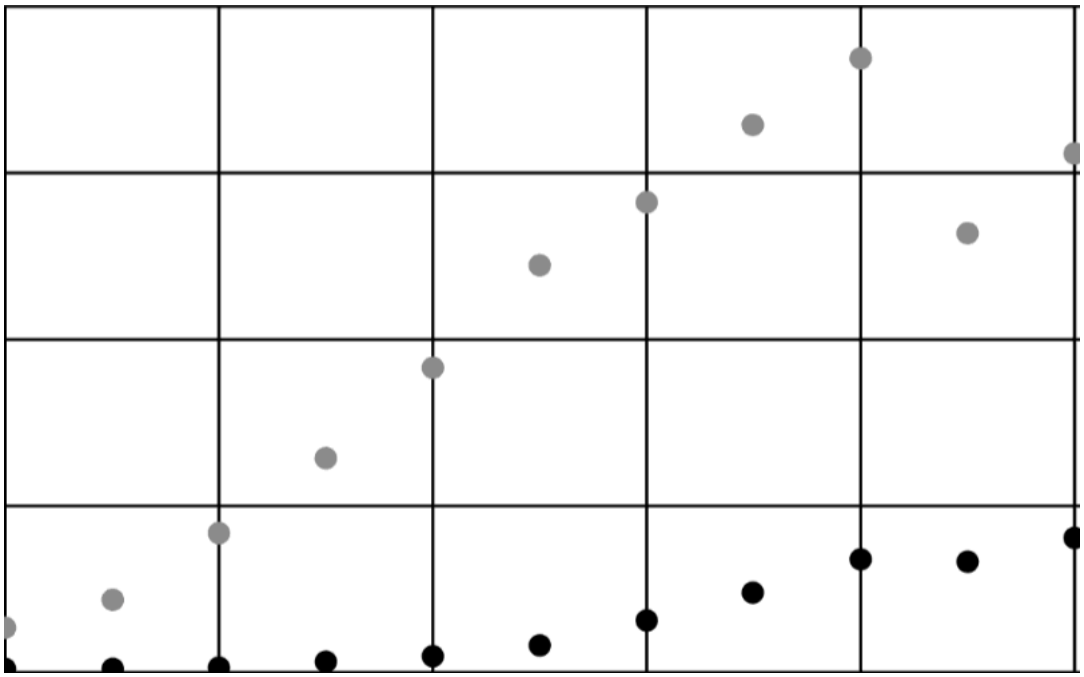
SSD와 HDD의 SSD 랜덤접근성능 비교 (지원기능on)

다음 파라미터로 데이터를 얻는다고 가정한다.

I/O 지원기능	종류	패턴	1회당 I/O size
on	r	rand	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096
on	w	rand	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096

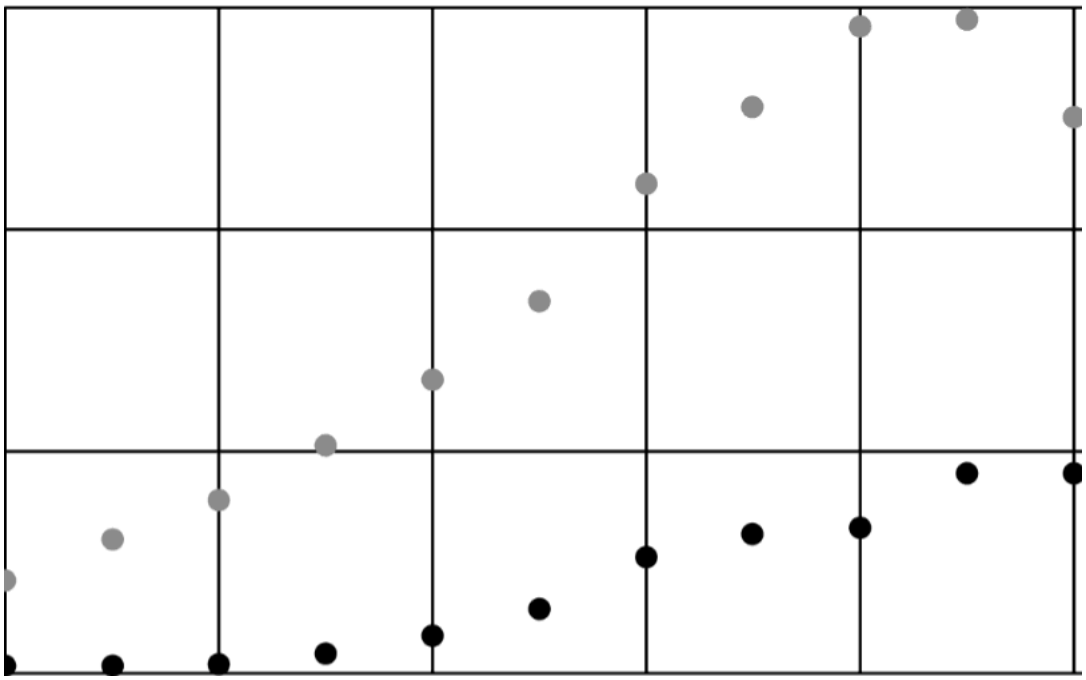
SSD와 HDD의 랜덤 읽기 성능(지원기능 on)

(검은점은 HDD, 회색점은 SDD)



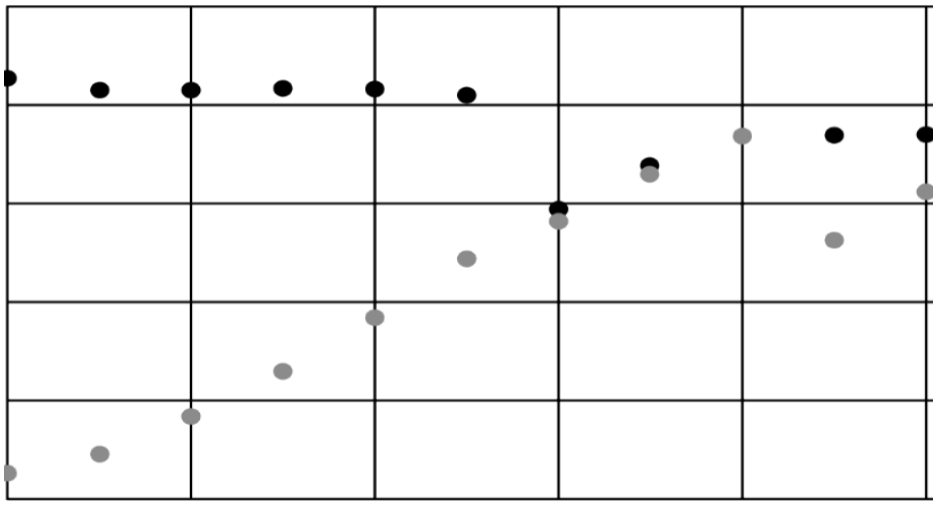
SSD와 HDD의 랜덤 쓰기 성능(지원기능 on)

(검은점은 HDD, 회색점은 SDD)

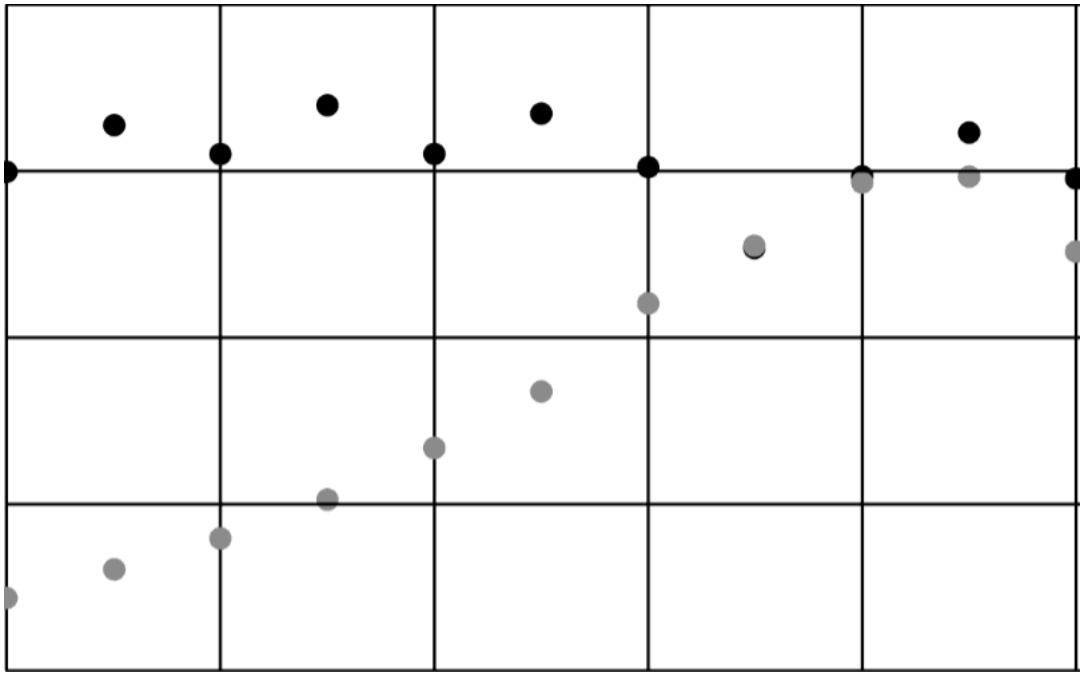


읽기와 쓰기 어떤 경우라도 I/O size가 커질수록 성능이 좋아지며 SSD와 HDD 둘중에는 SSD쪽이 성능이 좋음을 알 수 있다. 이제 시퀀셜 접근과 랜덤접근을 비교해보자.

SSD 읽기성능(지원기능on) (검은점 = 시퀀셜, 회색점 = 랜덤)



SSD 쓰기성능(지원기능on) (검은점 = 시퀀셜, 회색점 = 랜덤)

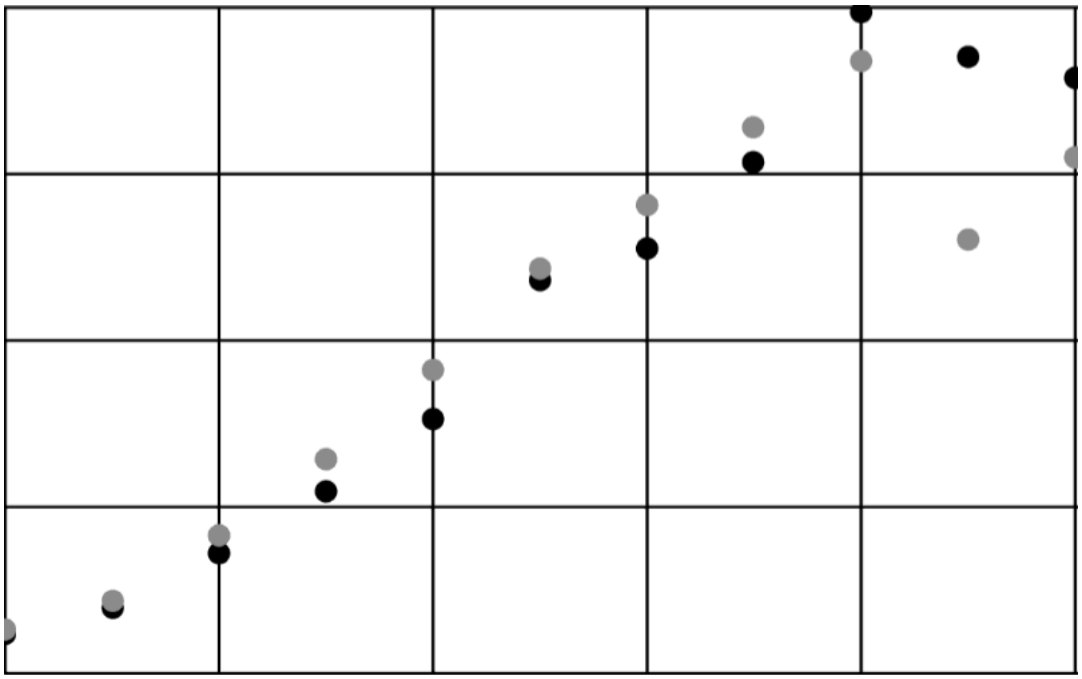


랜덤접근의 경우에는 I/O size가 작은 경우 시퀀셜 접근보다 훨씬 느리지만 어느정도 I/O size가 커지면 성능이 비슷해지는 것을 알 수 있다.

이제 마지막으로 I/O 지원 기능이 있는 경우와 없는 경우를 비교해보겠다.

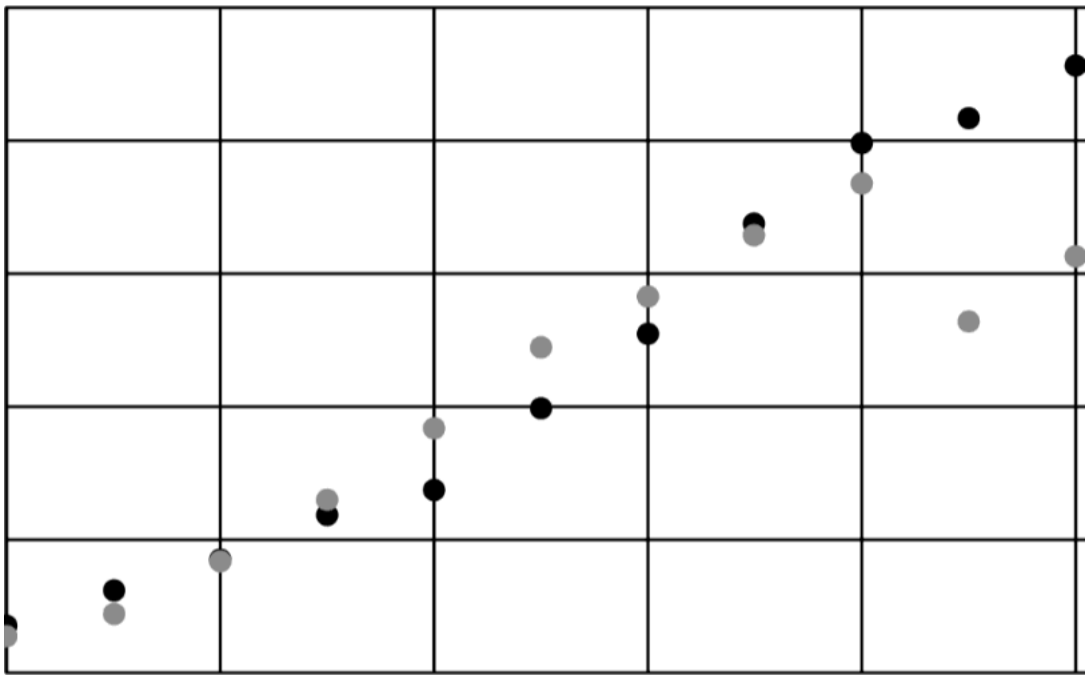
SSD 랜덤 읽기성능(지원기능on)

(검은점 =지원기능off, 회색점 = 지원기능on)



SSD 랜덤 쓰기성능(지원기능on)

(검은점 =지원기능off, 회색점 = 지원기능on)



읽기에 대해서는 별로 차이가 없어보인다. HDD의 경우와 마찬가지로 미리 읽기도 I/O스케줄러도 동작하지 않기 때문이다. 쓰기에 대해서는 I/O size가 작을 때는 효과가 있지만 금방 I/O지원 기능이 없는 경우보다도 성능이 떨어질 때가 있음을 알 수 있다. I/O스케줄러를 위해 여러 개의 I/O요청을 모아서 처리하는 오버헤드 SSD의 경우에는 무시할 수 없는 점과 정렬의 효과가 별로 없기 때문이다.

테스트한 결과를 보면, HDD의 경우나 SSD의 경우도 사용자가 의식하지 않아도 커널의 지원 기능 덕분에 어느정도 최적화가 되고 있음을 알 수 있었다. 그러나 모든 경우에 있어 최적의 성능이 나오지는 않았다. 특히 SSD는 특정 상황에서 I/O스케줄러가 오히려 성능을 떨어뜨리는 원인이 될 수 있음을 알 수 있었다.