2015211372 한준호

wnsgh8589@naver.com

요약

Linux 환경에서 벤치마크를 활용한  
Delay Allocation On/Off 상태에서의 SSD I/O 성능 테스트

**The effectiveness of delayed allocation on Ext4 file system**

목차

[1. 서론 2](#_Toc36923085)

[1.1 파일 시스템 2](#_Toc36923086)

[1.2 Ext4 3](#_Toc36923087)

[1.3 지연 할당 (Delay Allocation) 5](#_Toc36923088)

[2. 기본 설정 6](#_Toc36923089)

[2.1 사양 6](#_Toc36923090)

[2.2 테스트 환경 7](#_Toc36923091)

[2.3 파티션 8](#_Toc36923092)

[2.4 포맷 11](#_Toc36923093)

[2.5 마운트 12](#_Toc36923094)

[2.6 지연 할당 옵션 14](#_Toc36923095)

[3. Fio 15](#_Toc36923096)

[3.1 설치 15](#_Toc36923097)

[3.2 실행 방법 15](#_Toc36923098)

[3.4 iodepth 변경 18](#_Toc36923099)

[3.5 fsync 옵션 부여 22](#_Toc36923100)

[4. File bench 24](#_Toc36923101)

[4.1 설치 24](#_Toc36923102)

[4.2 실행 방법 25](#_Toc36923103)

[4.3 실행 결과 25](#_Toc36923104)

[5. YCSB 26](#_Toc36923105)

[5.1 설치 26](#_Toc36923106)

[5.2 실행 방법 28](#_Toc36923107)

[5.3 실행 결과 29](#_Toc36923108)

[6. 참고 자료 34](#_Toc36923109)

[6.1 사이트 34](#_Toc36923110)

[6.2 논문 34](#_Toc36923111)

1. 서론

* 1. 파일 시스템

파일 시스템이란 컴퓨터에서 데이터들을 쉽게 저장하고 찾고 유지 및 관리를 해주는 체계를 말합니다. 파일시스템은 하드디스크, CD-ROM과 같은 물리적인 저장 장치만 해당되지 않고 서버, 클라이언트와 같이 가상 형태의 접근 방식을 가진 저장 장치도 그 범위에 포함된다. 예전에는 데이터의 양도 적고 많은 데이터 처리가 필요가 없어 파일 시스템이 필요가 없었지만 요즘 대용량 저장매체들이 보편화되고 그만큼 많은 양의 데이터를 다루게 되면서 운영체제에서 이러한 많은 양의 데이터들을 관리하기 위해 많은 자원들을 사용하게 되었고, 이로 인해 다른 작업들을 하는데 어려움이 생기게 되었고 효율적인 데이터 관리를 위해 파일 시스템이 필요하게 되었습니다.

파일 시스템은 기본적으로 그림 1과 같이 메타 데이터 영역과 데이터 영역의 두가지 영역을 가진 구조로 표현이 됩니다. 이러한 구조를 활용하여 사용자는 빠르게 데이터 정보를 가져올 수 있습니다. 메타 데이터 영역은 데이터 영역에 기록된 파일의 이름, 위치, 크기, 시간정보 등등 데이터의 정보들을 저장하는 곳입니다. 데이터 영역에 기록된 파일 데이터를 직접 필요한 경우가 아니라면 메타 데이터 영역을 활용하여 빠르게 접근하고 해당 파일 정보들을 확인할 수 있습니다. 이러한 파일 시스템의 구조를 활용하면 파일을 실행할 때에 윈도우에서 탐색기를 통해 메타 데이터 영역을 탐색하고 해당 파일의 확장자에 맞는 응용 프로그램으로 파일을 실행합니다. 그리고 그 응용 프로그램은 해당 파일이 위치한 데이터 영역에서 파일을 읽어서 처리를 합니다. 데이터 영역만 존재하지 않고 메타 데이터 영역을 따로 구분하여 사용하는 만큼 데이터 영역이 줄어 들긴 하지만 이러한 구조의 장점으로 큰 효율을 볼 수 있습니다.

Window에서는 FAT와 NTFS, Linux에서는 Ext, Apple Mac에서는 HFS, Unix에서는 UFS, CD/DVD에서는 CDFS처럼 사용하는 OS 계열에 따라 각 OS에 맞는 다른 파일 시스템을 사용합니다. 본 실험에서 사용할 파일시스템은 Linux에서 사용하는 Ext 파일 시스템입니다.

|  |
| --- |
|  |
| [그림 1] 파일 시스템 기본 구조 |
|  |
|  |

* 1. Ext4

Ext4 파일 시스템은 현재 리눅스 커널의 기본 파일 시스템입니다. 최대 1EB의 볼륨과 16TB의 파일을 지원하며 64비트의 기억 공간 제한을 없애고 이전의 ext3의 단점을 보완하여 하위 호환성이 있는 확장 버전으로 ext3 사용자에게는 영향을 주지 않으면서 ext3에서 fork하여 ext4로 이름을 변경하여 제안되어 개발되었습니다. ext2와 ext3에 대한 하위 호환성으로 ext2와 ext3 파일 시스템을 ext4로 마운트 하는 것이 가능하고 ext4의 블록 할당 알고리즘을 기존의 파일 시스템에서 사용할 수 있습니다. 하지만 ext4의 파티션이 Extent를 사용한다면 ext3로 마운트가 불가능합니다.

Ext4 파일 시스템은 디스크의 블록들의 읽기 쓰기 성능을 향상시키기 위해 Extent, Multiblock Allocation, Delay Allocation과 같은 정책들을 추가했습니다. 기존의 ext3 파일 시스템에 경우 요청된 데이터를 처리할 때 블록 단위로 하는 블록 매핑 정책을 사용했습니다.

Ext4에서는 기존의 블록 매핑 기법을 대체하기위해 Extent 정책을 사용했습니다. [그림 2]와 [그림 3]을 보면 ext2와 ext3에서는 각각의 블록들의 트랙을 유지하기 위해 간접 블록 매핑을 사용했습니다. 연속적으로 블록을 할당하였다고 해도 각 데이터 블록을 가리키기 위해 inode에 존재하는 12개의 직접 블록과 3개의 간접블록을 이용하여 하나씩 지정해야 했습니다. 하지만 extent 기반 블록 유지방법은 간단한 정보만을 저장하여 메타 데이터의 크기를 절약하였고 데이터 블록의 접근에 걸리는 시간을 단축했습니다. Extent는 연속한 물리적 블록의 묶음을 의미하는데 초기에 어느정도 연속된 공간을 할당하고 그 공간이 부족할 경우 Extent의 단위로 할당을 하는데 최대 128MiB만큼의 연속된 블록에 할당할 수 있어서 한번에 처리할 수 있는 블록의 범위가 늘게 되었고 블록 매핑에서 생긴 단편화를 방지하고 대용량 파일처리시 좋은 I/O 성능을 나타냅니다.

블록 매핑 기법에 경우 ext3에서는 한번에 한 개의 블록(4KB)을 할당할 수 있는데 이러한 경우 용량에 비해 매핑 되는 블록의 단위가 작기 때문에 여러 번 매핑을 해야 했습니다. 그로 인해 성능이 크게 저하되었습니다. 예를 들어 100MB의 작은 용량이 필요한 경우에도 25600번(100MB = 102400KB, 102400/4 = 25600) 블록 할당을 하게 되는데 이렇게 작은 용량도 25600번의 매핑이 필요하다면 대용량의 데이터의 경우 더 많은 횟수의 블록 할당 호출이 발생하게 되고 연속된 블록으로 할당되지 않을 수 있어 단편화가 발생하게 됩니다. 단편화는 하나의 데이터를 한번에 연속적인 블록에 저장하지 못하고 여러 개의 불연속적으로 저장되는 상태를 말하는데 이런 겨우 데이터를 처리할 때 큰 성능 저하를 발생시킵니다. Ext4에서는 단일 블록 할당으로 인한 오버헤드를 막기위해 다중 블록 할당자를 사용했습니다. 매 호출마다 한번에 한 개의 블록을 할당하는 것이 아닌 한번에 여러 개의 블록을 할당하여 블록 할당의 횟수가 줄어들게 되었고 연속적인 블록에 할당하게 되어 오버헤드를 줄일 수 있게 되었습니다.

|  |
| --- |
| 텍스트, 지도이(가) 표시된 사진  자동 생성된 설명 |
| [그림 2] Ext3의 블록 매핑 |
|  |

|  |
| --- |
| 스크린샷이(가) 표시된 사진  자동 생성된 설명 |
| [그림 3] Ext4의 Extent |

* 1. 지연 할당 (Delay Allocation)

Ext4에 delay allocation 또는 allocate on flush라는 새로운 블록 할당 기법을 사용하게 되었습니다. 기존의 Ext3 파일시스템에서는 write() 함수가 호출되면 실제로 디스크에 데이터가 쓰여 지고 있지 않더라도 디스크에 그 데이터에 대한 블록을 할당했습니다. 하지만 만약 파일 크기가 고정되어 있지 않고 용량이 커지는 파일에 대해서는 지속적으로 write() 함수가 호출이 되고 이 경우 CPU의 사용량을 늘리게 되고 연속되지 않은 블록에 할당이 되어 큰 파일에 대해 단편화가 발생하게 되었습니다. 이러한 문제로 Ext4에서 지연 할당 기법을 사용하게 되었습니다. 파일이 캐시에 유지 되어있는 동안에는 그 파일이 실제로 디스크에 쓰일 때까지 블록 할당을 최대한 지연시킵니다. 이로 인해 캐시에 있는 파일 중 같은 파일에 대해서는 다중 블록 할당을 통해 인접한 공간에 할당하게 되었고 내부 단편화를 방지할 수 있게 되었습니다. Delay allocation의 경우 버퍼 캐시에 연속된 데이터들이 저장되면 Flush operation을 통해 디스크에 블록을 할당하여 연속된 블록에 할당을 하게 됩니다. Flush operation은 캐시를 flush하는 과정을 말하는데 캐시에 유지 되어있는 데이터들을 디스크에 저장하는 것을 의미합니다. Flush operation에는 fsync 함수와 fdatasync가 있습니다. fsync는 파일에 대한 데이터 변경이 일어날 시 호출되어 데이터와 메타데이터, 파일의 모든 변경점을 디스크에 기록합니다. fdatasync는 fsync와 같은 역할을 하지만 호출시 데이터만 저장하여 메타데이터는 디스크에 반영하지 않습니다.

연속된 파일을 할당한다는 가정하에 [그림 3]을 보면 3번의 write() 함수의 호출이 있었는데 지연 할당이 없는 경우 write() 호출할 때마다 디스크에 할당을 하게 되어 저장 장치에 불연속적으로 할당을 하게 되는데 지연 할당의 경우 write() 호출에도 디스크에 할당을 지연하다가 캐시를 flush 할 때에 디스크에 한번에 할당하게 되고 연속된 블록에 할당할 수 있게 됩니다.

|  |
| --- |
| 스크린샷이(가) 표시된 사진  자동 생성된 설명 |
| [그림 4] 지연 할당 유/무에 따른 디스크 할당 |

2. 기본 설정

2.1 사양

Ext4 파일시스템에서 SSD의 Delay Allocation에 따른 I/O 성능을 분석하기 위해 Intel i7-8700, DRAM 32GB, Samsung SSD 860 Pro(256GB)를 사용하였고 OS는 Ubuntu 19.10 버전을 사용하였고 Fio, File bench 벤치마크를 사용하여 정확한 수치 비교를 하였습니다.

|  |  |
| --- | --- |
| [표 1] 컴퓨터 사양 | |
| **CPU** | Intel i7 - 8700 (6 Cores 12 threads , 3.2GHz) |
| **Memory** | Samsung DDR4 8G \* 4 (32GB) |
| **Storage** | Samsung 860 PRO (256GB) |
| **OS** | Ubuntu 19.10 |
| **Kernel Version** | 5.3.0 - 23 |
| **File System** | Ext4 |
| **Benchmark** | Fio 3.12 / File bench 1.4.9 / YCSB 0.18.0 |

2.2 테스트 환경

1. 테스트에 사용할 SSD를 장착 후 컴퓨터 부팅
2. 터미널 실행
   * 리눅스에서는 명령어 입력을 통해 응용프로그램 실행과 세부 설정을 할 수 있기에

터미널 환경이 필요합니다.

* + [그림 5]와 같이 메뉴에서 터미널 아이콘 클릭하여 실행할 수 있습니다.
  + **Ctrl + Alt + T** 를 입력하여 터미널 아이콘 클릭없이 바로 실행할 수 있습니다.

(기존 터미널 은 자주색 배경으로 되어있는데 가독성을 위해 설정을 통해 색을 변경했습니다.)

|  |  |
| --- | --- |
|  | 스크린샷이(가) 표시된 사진  자동 생성된 설명 |
| [그림 5] 터미널 아이콘 | [그림 6] 터미널 화면 |

2.3 파티션

파티션은 하나의 디스크를 논리적으로 구분하기 위해 나누는 하나의 구획을 의미합니다. OS는 이렇게 나누어진 파티션을 별도의 드라이브로 인식합니다. 기본적으로 파티션은 백업과 같은 데이터 유지 및 보안상의 이유, 한 대의 PC에 여러 OS를 설치하기 위한 시스템의 안정성 확보, 개인적으로 파일의 구분 같이 관리의 편의성, 사용하는 OS의 디렉토리 특성 등 다양한 이유로 파티션을 하게 되었습니다.

테스트에서는 SSD를 Linux 환경에서 사용하는 ext4 파일 시스템을 이용하기 위해 파티션을 진행하였고 별도의 구분없이 전체를 사용하기 위해 한 개의 파티션만 생성하여 테스트를 진행하였습니다.

SSD의 파티션 지정을 위해 **fdisk** 명령어 사용 (**fdisk** – 파티션 테이블 관리 명령어)

1. **sudo fdisk -l** : 현재 PC에 연결된 스토리지들과 파티션을 리스트로 출력

* 현재 연결 된 디스크 중 해당 SSD 모델명, 남은 용량을 통해 테스트할 SSD 확인
* [그림 7]에서 볼 수 있듯이 연결된 SSD와 경로, 현재 파티션 상태를 확인 가능

1. 장착한 SSD의 **경로**를 확인

* [그림 7]과 같이 현재 디스크가 연결된 경로를 파악
  + ex) **/dev/sda**

|  |
| --- |
| 스크린샷, 대형, 사람들, 방이(가) 표시된 사진  자동 생성된 설명 |
| [그림 7] 연결된 SSD를 fdisk -l로 확인 |

1. **sudo fdisk (경로)** : (경로)에 연결된 SSD의 파티션 설정 커맨드 창 출력
   * **fdisk** 명령어를 통해 파티션 테이블 명령어 입력창을 출력
   * ex) **sudo fdisk /dev/sda**
2. **p**를 입력하여 현재 파티션이 존재하는지 확인
   * 불필요한 파티션이 존재할 경우 **d**를 입력하여 파티션을 제거

|  |
| --- |
|  |
| [그림 8] fdisk command 창에서 p입력을 통해 현재 파티션 확인 |

1. **n**을 입력하여 새로운 파티션 생성
   * enter를 누르며 넘어갈 경우 각 항목별 정해진 default 값이 자동 입력되어 생성
   * default 값은 1개의 파티션을 생성하고 전체 용량을 사용
2. **p**를 입력하여 파티션 생성을 확인 [그림 9]
   * ex) /dev/sda1 (sda의 첫번째 파티션)
3. **w**를 입력하여 저장하고 커맨드 창 종료
4. **f단계에서 p 명령어를 통해 파티션이 생성되지 않았을 경우**

* c단계부터 다시 실행
* 만약 파티션이 존재할 경우 **d** 명령어를 입력하여 파티션 정상 제거 후

파티션 생성 과정을 실행

|  |
| --- |
|  |
| [그림 9] fdisk command를 통해 생성한 파티션 **(/dev/sda1)** |

2.4 포맷

포맷을 하게 되면 그 저장장치에 있는 데이터들이 모두 삭제가 되어 흔히 저장 장치 초기화라는 의미로 알려져 있지만 원래 포맷은 처음 사용하는 디스크 또는 디스크의 재사용을 위해 자료를 저장할 수 있도록 형식을 잡아주는 것을 의미한다. 각각 사용하는 환경에 맞는 파일 시스템의 형식으로 포맷을 하여 사용하는데 Windows에서 사용할 경우 FAT이나 NTFS, Linux 환경에서는 Ext 파일 시스템으로 포맷을 하여 사용합니다.

테스트에서는 생성된 파티션을 사용하기 위해 Linux 환경에서 사용하는 ext4 파일 시스템으로 포맷하여 테스트를 진행합니다.

1. **mkfs.(포맷 형식) (경로)**
   * 현재 Linux 환경에 맞는 Ext4 형식으로 포맷
   * ex) **mkfs.ext4 /dev/sda1**
   * [그림 10]과 같은 화면이 나오면 Ext4 형식으로 포맷 완료

|  |
| --- |
|  |
| [그림 10] ext4 포맷 |

2.5 마운트

마운트는 디스크와 같은 물리적인 장치를 특정 위치(디렉토리)에 연결시켜 주는 것을 의미합니다. OS에따라 마운트 방법이 다른데 현재 쓰는 PC나 노트북에서 주로 사용하는 운영체제인 Windows에서는 USB, 외장 하드디스크, SSD 등등 저장 장치를 장착하는 것 만으로 인식이 되고 사용이 가능합니다. PnP(Plug and Play)라는 기능 덕분에 내부에서 자동으로 마운트 작업을 하여 사용자가 디스크를 편리하고 빠르게 사용할 수 있게 해줍니다. Linux의 경우 PnP 기능이 존재 하자지만 시스템 부팅 후 다시 마운트 하여 관리자가 직접 특정 디렉토리에 연결을 시켜줘야 사용이 가능하기도 합니다. 지속적으로 장착하여 사용하는 저장 장치의 경우 매번 마운트 하는 것이 번거로울 수가 있는데 명령어를 통해 자동 마운트 하는 방법도 있습니다.

테스트에서 ext4 파일 시스템으로 포맷한 SSD를 사용하기 위해 바탕화면에 만들어진 디렉토리에 마운트를 하여 SSD를 하나의 디렉토리처럼 사용합니다.

1. 마운트를 위해 마운트를 할 특정 위치(디렉토리)를 생성
   * 바탕화면에 SSD라는 폴더를 생성 (경로 : /home/hjh2020/Desktop/SSD or ~/Desktop/SSD)
   * **바탕화면의 경로 : /home/(계정 이름)/Desktop**
2. **sudo mount (SSD 경로) (마운트 할 경로)**

* 바탕화면에 생성된 SSD라는 폴더와 Ext4 포맷을 해준 파티션을 연결 해준다.
  + ex) **sudo mount /dev/sda1 ~/Desktop/SSD**
  + 마운트 해제 시 **umount 명령어 사용** / ex) **sudo umount (SSD경로)**

1. **sudo mount** 명령어를 입력하여 마운트 상태를 확인 [그림 11],[그림 12],[그림 13]
   * 마운트 성공 시 디렉토리 상태가 바탕화면의 SSD 폴더에서 SSD로 바뀐 것을 확인 가능

|  |
| --- |
| 스크린샷, 시계, 모니터이(가) 표시된 사진  자동 생성된 설명 |
| [그림 11] 마운트 시 디렉토리의 상태 변화 |

|  |
| --- |
|  |
| [그림 12] SSD 마운트 실행 후 마운트 명령어를 통해 확인 |
|  |
| [그림 13] 해당 스토리지 마운트 확인 |

2.6 지연 할당 옵션

Delay Allocation On/Off (지연 할당 유/무) 상태에 따라 SSD의 I/O 성능 비교를 위해

마운트 시 nodelalloc 옵션을 사용하여 Delay Allocation On/Off 상태를 설정

1. **sudo mount -o (옵션) (SSD 경로) (마운트 할 경로)**
   * ex) **sudo mount -o nodelalloc /dev/sdb1 ~/Desktop/SSD**
2. **sudo mount** 명령어를 입력하여 마운트 상태를 확인

* 옵션 유/무에 따라 상태의 변화 확인 가능 [그림 14]
* mount 옵션을 통해 현재 파티션의 Ext4 포맷상태도 확인 가능

|  |
| --- |
|  |
| [그림 14] 마운트 옵션 설정 후 상태 변화 |

3. Fio

SSD의 I/O 성능 테스트를 위해 Fio 벤치마크를 사용

3.1 설치

1. **sudo apt-get install fio** 명령어 입력 [그림 15]

|  |
| --- |
|  |
| [그림 15] fio 설치 화면 |

3.2 실행 방법

* 2가지 실행 방법이 존재
* 명령어를 통해 실험을 하는 방식과 fio 파일을 만들어 실행하는 방법이 존재

1. 명령어 입력 방식
   * **fio --(옵션)=(설정값)**
   * ex) **fio --directory=(마운트 한 폴더) –name=(파일 이름) --direct=(int) --rw=(방식) --bs=(블록 단위) --size=(파일 용량) --time\_based --runtime=(실행 시간(초)) --group\_reporting –norandommap**
2. fio 파일 생성 및 실행
   * 명령어를 통해 입력하지 않고 fio 파일을 생성하여 파일 안에 명령어 옵션들을 입력하고 실행
   * **fio (파일이름).fio**

|  |  |  |  |
| --- | --- | --- | --- |
| [표 2] fio 명령어 | | | |
| **name = [string]** | 테스트 이름 | **rw = [I/O 종류]** | 테스트 할 I/O 종류 |
| **filename = [name or path]** | 테스트 파일 or 경로 | **numjobs = [int]** | Job 수 |
| **directory = [dir 경로]** | 디렉토리 경로 | **ioengine = [I/O 방법]** | 테스트 할 I/O 방법 |
| **nrfile = [int]** | Job당 파일 수 | **ramp\_time = [int]** | 각 테스트 사이 여유 시간 |
| **size = [int]** | Job당 총 파일 사이즈 | **iodepth = [int]** | 큐의 깊이 |
| **bs = [int]** | Block의 사이즈 | **runtime = [int]** | 테스트 진행 시간 |
| **direct = [bool]** | True = Direct I/o  False = Bufferd I/O | **norandommap** | 랜덤 워크로드시 과거 I/O위치 고려안함 |
| **group\_reporting** | Job이 아닌 그룹별 리포트 | **stonewall** | 직전 테스트 종료 후 다음 테스트 진행 |

|  |  |
| --- | --- |
| [표 3] fio 파일 예시 | |
| [global]  direct=1  ioengine=libaio **[global] 테스트에 공통으로**  ramp\_time=60 **사용되는 파라미터**  size=40G  iodepth=1  runtime=180  time\_based  group\_reporting  norandommap  [read-4k-para]  bs=4k  stonewall  filename=/home/hjh2019/Desktop/SSD/rtest  rw=read | [write-4k-para]  bs=4k  stonewall  filename=/home/hjh2019/Desktop/SSD/wtest  rw=write  [randread-4k-para]  bs=4k  stonewall  filename=/home/hjh2019/Desktop/SSD/rrtest  rw=randread  [randwrite-4k-para]  bs=4k  stonewall  filename=/home/hjh2019/Desktop/SSD/rwtest  rw=randwrite |

3.3 실행 결과

Delay Allocation On 상태에서 순차 읽기 쓰기 / 랜덤 읽기 쓰기 Delay Allocation Off 상태에서 순차 읽기 쓰기 / 랜덤 읽기 쓰기 테스트를 진행하고 실험 결과를 text 파일로 저장하여 그 수치들을 그래프로 표현

* iops \* 블록단위 = 초당 데이터 전송량
* Delay Allocation On/Off 상태에 성능 차이 미비

지연 할당 유무 상태에서 실험을 진행하였으나 명확한 성능 변화가 보이지 않아 옵션을 변경하여 재실험을 하였습니다.

3.4 iodepth 변경

Iodepth 옵션을 변경하여 큐의 깊이를 다르게 하여 추가 실험을 하였습니다.

* iodepth 1
* iodepth 2
* iodepth 4
* iodepth 8
* iodepth 16
* iodepth 32
* iodepth 64
* iodepth 변화에 의한 Delay Allocation On/Off 상태에 성능 변화 미비

iodepth 옵션을 변경하여 큐의 깊이를 다르게 하여 재실험을 하였지만 순차/랜덤 읽기 쓰기에 대한 성능 차이가 미비 하여 다른 옵션을 추가로 재실험을 하였습니다.

3.5 fsync 옵션 부여

저장 장치의 I/O 연산은 커널 안의 버퍼 캐시나 페이지 캐시를 거쳐서 이루어집니다. 캐시(Cache)란 저장 장치와 CPU 사이에 속도 차이를 줄여 주기 위한 작은 메모리 공간입니다. 지연 할당에 경우 데이터가 쓰일 때까지 최대한 지연시켜 flush operation이 실행될 때 블록을 할당하게 되는데 fsync는 그 flush operation의 함수 중 하나로 파일에 대한 데이터의 변경이 발생하였을 경우 그 데이터와 그 데이터의 메타데이터 모두를 디스크에 저장을 합니다.

테스트를 통해 결과 값이 나오지 않아 새로 fsync 옵션을 부여하여 테스트를 진행하였습니다. 지연 할당의 경우 데이터가 쓰일 때 지연 할당이 되므로 읽기의 과정은 없애고 fsync옵션에 따라 Delay Allocation On/Off를 하여 순차 쓰기 / 랜덤 쓰기 테스트를 진행하였습니다.

* fsync 옵션의 default 값 = 0
  + fsync = 1로 설정 했을 때
  + fsync = 1로 테스트 후 iodepth의 변화를 주었지만 성능 차이 미비

fsync = 0의 경우 iops의 성능이 크게 감소하였습니다.

* + 파일의 개수를 늘리고 fsync = 100을 주고 재실험
* fsync = 100 / 1GB의 파일 10개

fsync옵션을 주어 1GB의 파일 10개로 실험을 하였으나 지연 할당에 따른 성능 차이 미비

fsync옵션에 대한 변경을 하고 재실험

4. File bench

4.1 설치

1. 홈페이지를 통해 다운로드 (tar 압축 파일)

* <https://sourceforge.net/projects/filebench/>

1. 다운로드 경로에 들어가서 압축 풀기

* **cd** 명령어를 통해 Tar 압축 파일이 다운로드 된 경로 진입
* **cd** – change directory (디렉토리 변경)
  + **tar -xvf (파일명).tar** 명령어를 통해 압축 해제
  + ex) tar -xvf filebench.tar

1. 해당 폴더에 진입
   * **cd** 명령어 사용
   * ex) cd filebench
2. **./configure** 명령어로 configure 파일 실행
3. **sudo make**
4. **sudo make install**
5. **filebench** 입력하여 정상 설치 확인 [그림 16]

|  |
| --- |
|  |
| [그림 16] filebench 정상 설치 확인 |

4.2 실행 방법

1. 설치된 filebench/workload 디렉토리에 fileserver.f, varmail.f, webserver.f 파일을 다른 곳으로 복사
2. 메모장을 통해 각 파일들을 열어서 설정 값 입력
3. **filebench -f (파일이름).f** 명령어로 실행
   1. ex) **filebench -f fileserver.f / filebench -f varmail.f / filebench -f webserver.f**

4.3 실행 결과

* Delay Allocation On/Off에서 iops를 측정한 그래프
* 데이터 전송량 / 60 / 1024 = 초당 데이터 전송 량(MB단위)

5. YCSB

야후에서 만든 클라우드 서비스의 성능을 측정하기 위한 벤치마크 툴로 특정 상황에서 NoSQL을 도입할 때 해당 NoSQL이 사용에 적합한지 또는 여러 개의 NoSQL중 어떤 것이 가장 적합한지 상황에 맞는 벤치마킹 환경 및 결과를 제공합니다. Java를 기반으로 작성된 테스트로, 패키지 관리는 maven으로 작성되어 있습니다. 지원하는 NoSQL이 많고 Workload를 사용하는 것이 간단합니다. 기본 Workload가 대부분의 기능을 지원하기 때문에 특정 상황에 맞게 쉽게 시뮬레이션 할 수 있습니다. 각 Workload에서 지원하는 인터페이스는 5개로 init(초기화), insert(삽입), update(갱신), read(읽기), delete(삭제)가 있습니다. 5개를 활용한 Workload는 총 6개로 A부터 F로 나뉘어져 있습니다. A는 업데이트 중심의 작업으로 읽기50%, 업데이트 50%를 수행하고 B는 읽기 중심의 작업으로 읽기 95%, 업데이트 5%로 한번의 태그 작성을 하고 주로 읽기 작업을 수행합니다. C는 읽기 전용 작업으로 읽기 100%를 수행하고 D는 최근 레코드 중심의 읽기 작업으로 읽기 95%, 쓰기 5%로 최근 저장된 레코드를 중심으로 읽기를 수행합니다. E는 영역 스캔으로 읽기 95%, 쓰기 5%로 각 작업마다 100개 내외의 레코드 영역을 한 번에 쿼리를 합니다. F는 읽기-쓰기-수정으로 읽기, 수정, 쓰기 작업을 순서대로 수행합니다

5.1 설치

1. Github을 통해 압축 파일을 다운받기

* (<https://github.com/brianfrankcooper/YCSB>)

1. 다운로드 폴더에서 ZIP 파일을 압축 해제
   * tar 파일인 경우 **tar -xvf (파일명).tar** 명령어를 통해 압축 해제
2. 터미널 창에서 cd 명령어를 통해 압축 해제된 YCSB 폴더 진입
   * **cd YCSB**
3. **mvn** 명령어를 통해 패키지 설치
   * **mvn clean package** (YCSB안에 전체 패키지 설치) [그림 18]
   * **mvn -pl site.ycsb:[패키지 이름]-binding -am clean package** (해당 패키지 한 개만 설치)

**주의사항**

* maven 2 버전 사용시 <https://github.com/brianfrankcooper/YCSB/issues/406> 에러 발생
* **maven 3 버전 설치 필수 [그림 17]**
  + 만약 maven 미설치시 **sudo apt-get install maven** 명령어를 통해 maven 최신 버전 설치

|  |
| --- |
| 스크린샷이(가) 표시된 사진  자동 생성된 설명 |
| [그림 17] maven 미설치시, 업데이트 및 버전 확인 |

1. Build Success로 패키지 설치 확인 [그림 19]

|  |
| --- |
| 스크린샷이(가) 표시된 사진  자동 생성된 설명 |
| [그림 18] mvn clean package (전체 패키지 설치) |
| 스크린샷, 컴퓨터이(가) 표시된 사진  자동 생성된 설명 |
| [그림 19] 설치 완료시 Build Success 확인 |

5.2 실행 방법

1. YCSB 폴더 내에서 **./bin/ycsb** 명령어를 통해 실행

1. 진행하려 하는 rocksdb에 워크로드 불러오기 워크로드 실행 순으로 진행
   * **./ycsb load rocksdb -s -P workloads/[워크로드 종류] -p rocksdb.dir=[실행 시킬 디렉토리]**
   * **./ycsb run rocksdb -s -P workloads/[워크로드 종류] -p rocksdb.dir=[실행 시킬 디렉토리]**
2. 해당 디렉토리 내에 생성된 폴더에서 log파일에서 값 확인

|  |
| --- |
| 스크린샷, 테이블, 노트북, 오렌지이(가) 표시된 사진  자동 생성된 설명 |
| [그림 20] ./bin/ycsb 명령어로 설치 확인 및 기본 사용법 확인 |
|  |
| [그림 21] rocksdb에 워크로드를 불러오기 |
|  |
| [그림 22] rocksdb에 워크로드 실행하기 |

5.3 실행 결과

1. YCSB에서 기본으로 제공하는 Workload F를 이용하여 Delay Allocation On/Off 상태에서의 성능 비교를 진행, 생성된 log파일 내에서 결과값 확인

기본 Workload F를 이용하여 YCSB 벤치마크 실행한 후 log파일의 확인결과

* Delay allocation On에서 151.2 MB/s Delay allocation Off에서 92.4 MB/s의 결과 값이 확인

1. 각 Workload마다 Record Count와 Operation Count를 50,000,000으로 설정하여 Delay Allocation On/Off 상태에서의 비교 실험 진행

Workload a

* + - Workload a 실험 결과 Runtime의 경우 Delay Allocation On시 Delay Allocation Off보다 16분 더 빠르게 끝났으며 처리량(ops/sec)은 1741정도 더 많음
    - Workload a의 경우 Delay Allocation On시 성능이 더 좋음

Workload b

* Workload b 실험 결과 Runtime 모두 55분에 끝났으며 처리량(ops/sec)은 Delay Allocation On시 Delay Allocation Off보다 1941정도 더 적음
* Workload b의 경우 Delay Allocation On시 성능이 더 안 좋음

Workload c

* Workload c 실험 결과 Runtime의 경우 Delay Allocation On시 Delay Allocation Off보다 3분 더 느리게 끝났으며 처리량(ops/sec)은 839정도 더 적음
* Workload c의 경우 Delay Allocation On시 성능이 더 안 좋음

Workload d

* Workload d 실험 결과 Runtime의 경우 Delay Allocation On시 Delay Allocation Off보다 8분 더 느리게 끝났으며 처리량(ops/sec)은 5933정도 더 적음
* Workload d의 경우 Delay Allocation On시 성능이 더 안 좋음

Workload e

* Workload e 실험 결과 Runtime의 경우 Delay Allocation On시 Delay Allocation Off보다 16분 더 빠르게 끝났으며 처리량(ops/sec)은 125정도 더 많음
* Workload e의 경우 Delay Allocation On시 성능이 더 좋음

Workload f

* Workload f 실험 결과 Runtime의 경우 Delay Allocation On시 Delay Allocation Off보다 2분 더 느리게 끝났으며 처리량(ops/sec)는 229정도 더 적음
* Workload f의 경우 Delay Allocation On 시 성능이 더 안 좋음

1. Workload a의 Record Count와 Operation Count를 150,000,000로 설정하여Runtime을 늘려서 실험

* Workload a 실험 결과 Runtime의 경우 Delay Allocation On시 Delay Allocation Off보다 24분 더 빠르게 끝났으며 처리량(ops/sec)는 373 정도 더 많음
* Workload a의 경우 Delay Allocation On시 성능이 더 좋음

6. 참고 자료

6.1 사이트

* + <https://forums.servethehome.com/index.php?threads/benchmarking-with-fio-and-generating-graphs.7665/>
  + <https://blog.soobinpark.com/142>
  + http://forensic.korea.ac.kr/DFWIKI/index.php/%ED%8C%8C%EC%9D%BC%EC%8B%9C%EC%8A%A4%ED%85%9C\_%EA%B5%AC%EC%A1%B0
  + https://harryp.tistory.com/663
  + https://harryp.tistory.com/664
  + https://zetawiki.com/wiki/%EB%A6%AC%EB%88%85%EC%8A%A4\_fio
  + <https://fio.readthedocs.io/en/latest/fio_doc.html>
  + <https://m.blog.naver.com/PostView.nhn?blogId=gksgus092&logNo=221080447647&proxyReferer=https%3A%2F%2Fwww.google.com%2F>
  + <https://dev-ahn.tistory.com/98>
  + <https://experiences.tistory.com/11>
  + <https://en.wikipedia.org/wiki/Allocate-on-flush>
  + <https://songsunbi.tistory.com/62>
  + <https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload>
  + <https://crazia.tistory.com/entry/YCSB-%EC%95%BC%ED%9B%84-%ED%81%B4%EB%9D%BC%EC%9A%B0%EB%93%9C-%EC%84%9C%EB%B9%99-%EB%B2%A4%EC%B9%98%EB%A7%88%ED%81%AC-Yahoo-Cloud-Serving-Benchmark>

6.2 논문

* + Filebench 벤치마크 i/o 특성 분석
  + Ext4 파일시스템에서의 지연할당 성능 분석
  + 가상화 시스템에서의 파일시스템 블록 할당 기법 효용성 분석