2014211523강승욱

컴퓨터 종합 설계

BtrFS 성능 실험

컴퓨터 종합 설계

[1. BtrFS 2](#_Toc38359374)

[**1-1.** **CoW (Copy-on-Write)** 2](#_Toc38359375)

[**1-2.** **BtrFS의 기능** 2](#_Toc38359376)

[**1-3.** **BtrFS의 구조** 3](#_Toc38359377)

[**1-4.** **Ext4와 다른점** 4](#_Toc38359378)

[**1-5.** **BtrFS 옵션** 4](#_Toc38359379)

[2. 카산드라 (Cassandra) 6](#_Toc38359380)

[3. Btrfs, ext4 비교 fio 실험 7](#_Toc38359381)

[**3-1. 실험 전 설정하기** 7](#_Toc38359382)

[**3-2. 벤치마크 fio** 10](#_Toc38359383)

[**3-3 실험 환경** 11](#_Toc38359384)

[**3-4 실험** 12](#_Toc38359385)

[**3-5 실험 결과** 13](#_Toc38359386)

# **BtrFS**

B-tree file system 또는 Butter file system이라 부르며 이 파일시스템은 내결함성, 복구 및 쉬운 관리에 중점을 두고 고급 기능을 구현하기 위한 리눅스 용 CoW(Copy-on-Write) 기술을 사용하는 파일 시스템입니다. 기존 파일 시스템에 비해 데이터 관리 및 안정성 확보를 강화한 방식의 파일 시스템입니다. Linux 파일 시스템에서 풀링, 스냅샷, 체크섬 및 통합다중장치의 부족 문제를 해결합니다.

## **CoW (Copy-on-Write)**

중복 또는 복사 작업을 효율적으로 구현하기 위해 사용되는 리소스 관리 기술입니다. 여러 작업이 동일한 데이터를 사용하는 경우 리소스를 처리하고 컬렉션의 복사본을 유지 관리하는 최적화 기술입니다. BtrFS는 모든 파일에 대해 항상 CoW를 사용합니다. CoW 정책으로 블록을 업데이트 할 때 데이터 블록을 메모리로 읽고 수정한 다음 새 위치에 기록하여 이전 데이터를 수정하지 않습니다. 데이터 보호를 위한 스냅 샷을 구현합니다.

## **BtrFS의 기능**

* 동적 inode 할당
* 기록 가능 스냅샷, 스냅샷에 대한 스냅샷
* 하위볼륨 (별도의 내부 파일 시스템 루트)
* 오브젝트 차원에서의 미러링 및 스트리핑
* 데이터 압축 : Zlib, LZO를 통한 압축 알고리즘 지원
* 온라인 및 오프라인 파일 시스템 검사
* 서브볼륨관리 : 각각의 서브볼륨을 파티션처럼 취급 가능
* SSD 최적화 모드
* 온라인 단편화 제거 등

## **BtrFS의 구조**

내부적으로 모든 자료들을 B tree를 사용하여 관리하며, 8개의 트리가 존재합니다.

1. 루트 트리 : 다른 모든 트리들의 루트를 가리키기 위한 정보를 저장하는 트리입니다.
2. 파일 시스템 트리 : 사용자가 볼 수 있는 파일들과 디렉터리들은 모두 파일 시스템 트리 내에 존재하며, 서브볼륨 하나당 하나의 파일 시스템 트리가 존재합니다. 파일 시스템 트리 내에서, 각 파일과 디렉토리 객체는 inode item을 갖으며 inode, dir 등과 같은 파일 관련 메타 데이터를 관리합니다.
3. 익스텐트 할당 트리 : 파일 시스템에 대한 할당 맵의 역할을 하고 사용 가능한 스토리지를 여러 개의 유연한 할당 정책으로 나눕니다. (디스크 공간 할당을 관리), Query시 사용 가능한 디스크 공간 정보를 제공합니다. 파일 데이터는 트리 외부의 디스크 블록들이 연속적으로 인접하는 익스텐트에 보관됩니다.
4. 체크섬 트리 : 데이터와 메타데이터를 위해 체크섬이 계산되고, 트리 내에 체크섬 아이템으로 저장합니다.
5. 로그 트리 : 자신의 익스텐트를 추적하고 자신의 체크섬 아이템들을 유지하면서 자립적입니다. 이 아이템들은 다음 전체 트리 커밋이나 다음에 다시 마운트 할 때, 재생되었다가 삭제됩니다.
6. 청크 트리 : 각 장치를 장치 아이템으로 표현, 논리적 청크로부터 청크맵 아이템 내에 저장되고 모든 청크들은 청크 트리로 추적이 가능합니다.
7. 장치 할당 트리 : 청크들로 할당되어진 각 물리적 장치의 부분들을 기록합니다. 새로운 청크의 할당으로 갱신되는 상대적으로 작은 트리입니다. BtrFS 볼륨이 포함하는 청크들을 찾아 내용을 재배치함으로 효율적인 축소나 장치 제거가 가능하도록 합니다.
8. 재배치트리 : 단편화 제거, 축소와 재균형 명령은 익스텐트들의 재배치를 필요로 합니다. 단순히 재배치할 익스텐트의 CoW를 하는 것은 스냅샷들 사이의 공유를 파기하고, 다스크 공간을 낭비합니다.

공유를 유지하기 위해 메타데이터에 대해 전용의 스크래치 공간으로 사용되는 재배치 트리와 함께 update-and-swap 알고리즘이 사용됩니다. 이것은 영향을 받는 서브볼륨의 파일 시스템 트리를 이용하여 이전 익스텐트를 가리키는 메타데이터는 계속해서 새 익스텐트를 가리키도록 갱신되고 갱신이 완료되면, 재배치 트리 내 항목들은 영향을 받는 서브볼륨에 대응 항목들과 교체되고, 트리는 버려집니다.

## **Ext4와 다른점**

|  |  |  |
| --- | --- | --- |
| 항목 | BtrFS | Ext4 |
| 도입 운영체제 | 리눅스 | 리눅스 |
| 디렉토리 구조 | B-tree | 링크드리스트, hashed B-tree |
| 파일 구조 | Extents | Extents, Bitmap |
| 최대 파일크기 | 16EiB | 16TiB (4k 블록 파일 시스템 기준) |
| 최대 파일수 | 2의 64승 | 40억 |
| 최대 볼륨크기 | 16EiB | 1EiB |
| 암호화 가능여부 | 예 | 아니오 |
| 압축 가능여부 | 예 (zlib, LZO, ZSTD) | 아니오 |

BtrFS 스냅샷 : BtrFS는 자신의 서브볼륨 구조를 이용해서 파일을 예전 상태로 되돌릴 수 있는 스냅샷 기능을 제공합니다. BtrFS 형식으로 포맷된 볼륨에서 원하는 디렉토리를 원하는 시점에 스냅샷으로 저장해두면 언제든지 저장했던 상태로 되돌릴 수 있고 디렉토리 전체 뿐만 아니라 특정 파일만 되돌리는 것도 가능합니다. 스냅샷 기능은 파일 자체를 백업하는 것이 아니라 파일의 변경 이력을 저장하는 방식을 사용하기 때문에 저장 공간도 많이 차지 하지 않고 디스크 부하도 적습니다. 즉 이전의 Ext4 파일 시스템과 달리 백업을 위한 두배의 저장소 공간을 필요로 하지 않기 때문에 저장소 공간을 효율적으로 확보할 수 있다.

## **BtrFS 옵션**

|  |  |  |
| --- | --- | --- |
| acl | Default=ON | ACL에 대한 지원 활성화/비활성화 |
| autodefrag | Default=OFF | 자동 파일조각 모음  (대규모데이터베이스 워크로드 적합하지 않다) |
| barrier | Default=ON | 장치에 전송된 보류 중 모든데이터와 메타데이터 블록을  동기화 후 다음 수퍼블록을 쓰고 다른 flush 실행 |
| check\_int | Default=OFF | 무결성 검사기 모듈 활성화/비활성화 |
| clear\_cashe |  | 문제가 발생한 경우 디스크 공간 캐시 지우고 다시 작성 |
| commit=seconds | Default=30 | 데이터가 영구 스토리지와 동기화 될 때,  주기적 트랜잭션 커밋 간격을 설정 |
| compress | Default=OFF | BtrFS 파일 데이터 압축 제어 |
| datacow | Default=ON | 새로 만든 파일에 COW 활성화/비활성화 |
| datasum | Default=ON | 새로 만든 파일에 데이터 체크섬 활성화/비활성화 |
| degraded | Default=OFF | RAID profile 제약 조건보다 적은 장치로 마운트 가능 |
| device=devicepath |  | 장치관리자 또는 btrfs장치 스캔 명령을 자동 수행 |
| discard | Default=OFF | 해제 된 파일 블록을 버리기 |
| enospc\_debug | Default=OFF | 일부 ENOSPC 조건에 대해 자세한 출력 사용 |
| fatal\_errors=action | Default=bug | 치명적인 오류 발생 시 취할 조치 (=bug, =panic) |
| flushoncommit | Default=OFF | 이전 트랜잭션에서 write로 인해 더러워진 데이터를 현재 커밋의 일부로 효과적으로 전체 파일 시스템 동기화로 커밋 (완료된 모든 파일시스템 작업 포함) |
| fragment=type | Default=OFF | 주어진 type의 블록 그룹을 의도적으로 fragment하는 디버깅 도우미 (디버깅 환경 외부에서 사용하면 안됨) |
| inode\_cache | Default=OFF | Free ionode 번호 캐싱 활성화/비활성화 |
| logreplay | Default=ON | 마운트시 로그 재생을 활성화/비활성화 (read 전용) |
| max\_inline=bytes | Default=min | 메타데이터 b-tree leaf에 inline될 수 있는 최대공간지정 |
| metadata\_ratio=value | Default=0 | 모든 value 데이터 청크 후 1개의 메타 데이터 청크가 할당되도록 지정 |
| norecovery | Default=OFF | 마운트시 데이토 복구를 시도하지 않기 |
| rescan\_uuid\_tree | Default=OFF | UUID tree의 강제 점검 및 재구축 절차 |
| skip\_balance | Default=OFF | 중단 된 balance operation 작동의 자동 재개를 skip함 |
| space\_cache | Default=v1 | 여유 공간 캐시를 제어하는 옵션 |
| ssd\_spread | Default=감지 | SSD 할당 체계를 제어하는 옵션, ssd 자동감지 |
| subvol=path |  | 최상위, 하위 볼륨이 아닌 path에서 하위 볼륨 마운트 |
| subvolid=subvolid |  | 지정된 파일 시스템에 대한 기본 하위 볼륨 세트를 대체 |
| thread\_pool=number | Default=min | 시작할 작업자 thread 수 |
| treelog | Default=ON | fsync 및 O\_SYNC write에 사용되는 트리로깅을 활성화 |
| usebackuproot | Default=OFF | 마운트시 잘못된 트리 루트가 발견되면 자동 복구 시도 |
| user\_subvol\_rm\_allowed | Default=OFF | 하위 볼륨을 해당 소유자가 삭제 가능 |

# **카산드라 (Cassandra)**

Java 언어 기반인 NoSQL 데이터베이스의 한 종류이며 아파치 오픈소스로 공개된 분산 데이터 베이스입니다. 대용량의 데이터 트랜잭션에 대해 고성능 처리가 가능합니다. 현재 Facebook을 포함하여 Twitter, Digg도 사용하고 또한 대량 데이터를 다루는 사이트에서 도입을 검토하고 있습니다.

대량의 데이터를 다루는데 효과적이고 확장성이 뛰어납니다. 다량의 데이터를 다수의 서버에 분산시키며 분산된 데이터를 여러 서버에 복제해 두는 방식으로 읽기, 쓰기가 뛰어납니다. 데이터를 메모리에 보관하고 일정 조건 후 디스크에 입력하는 영속성과 휘발성 key-value가 혼합된 방식으로 관계형 데이터베이스에 비해 속도가 빠릅니다. 칼럼 단위로 관리되므로 동일한 데이터라도 행단위로 관리하는 것보다 디스크에 많이 저장할 수 있습니다.

카산드라 장점

* 대량의 데이터를 분산된 서버에서 관리할 경우 적합합니다. 많은 사람들이 이용하고 많은 쓰기가 발생하는 서비스에 좋습니다.
* Consistent Hashing 방식을 이용하여 데이터를 분산시켜 대량의 요청이 서버에 몰리는 것을 막아 성능을 높일 수 있습니다.
* 신규 서버 생성시 중앙에 관리하는 서버가 없기 때문에 신규 서버만 기동하면 서비스에 추가됩니다.

카산드라 단점

* 복잡한 조건 검색을 못합니다. Row key, 칼럼을 인덱스 하기 때문에 검색 조건은 단순합니다.
* 데이터 갱신 및 입력 시 Atomic한 처리가 힘듭니다.

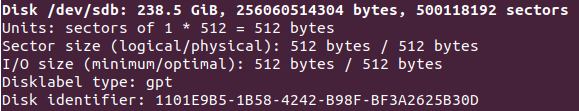
# **Btrfs, ext4 비교 fio 실험**

## **3-1. 실험 전 설정하기**

1. 실험에 사용할 SSD 장착합니다.
2. 우분투에서 터미널 실행합니다.
3. sudo fdisk -l 을 사용하여 디스크가 잘 장착되었는지 확인합니다.

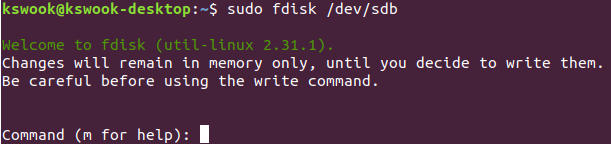


[그림1] fdisk list 명령어 작성



[그림2] fdisk list 명령어 결과 중 장착한 SSD표시

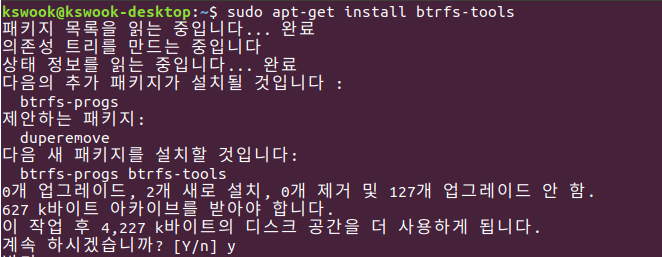
1. sudo fdisk /dev/sdb 을 사용하여 파티션을 분할합니다.



[그림3] fdisk 파티션 분할 명령어 입력

Command ( m for help ) : 가 나올 시, n 입력하기. 이때 파티션을 여러 개가 아닌 하나만 만들고 싶으면 다음 Command가 나올 때까지 Enter를 누릅니다. 이 Command에서 w를 눌러 저장합니다.

1. sudo apt-get install btrfs-tools 명령어 사용으로 btrfs를 사용할 수 있게 패키지 설치합니다. 계속 하시겠습니까? [Y/n]에서 y 입력합니다.



[그림4] btrfs 패키지 설치

1. sudo mkfs.(사용할 파일시스템 이름) /(경로) 사용할 파일 시스템으로 포맷해줍니다.



[그림5] btrfs 포맷

이때 btrfs 실험을 하기 위해 sudo mkfs.btrfs 를 입력후 그 뒤에는 위에서 파티션을 분할해주었던 경로 /dev/sdb1을 입력했다.

1. sudo mount /(경로)를 사용해 mount 할 위치에 mount줍니다.



[그림6] mount 명령어 작성

이때 /dev/sdb1 /mnt/test는 mount 할 경로로 작성합니다.

1. mount|grep /(경로)를 입력해서 mount 확인합니다.



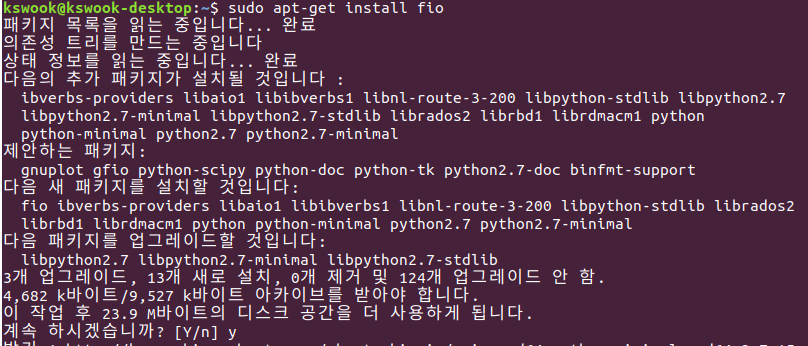
[그림7] mount 결과 확인 명령어 작성

이때 /dev/sdb1은 경로로 작성합니다.

## **3-2. 벤치마크 fio**

1. sudo apt-get install fio 명령어를 작성하여 설치합니다.

계속 하시겠습니까? [Y/n]에서 y 입력합니다.



[그림1] fio 패키지 설치

1. 벤치마크 fio 옵션

|  |  |
| --- | --- |
| 옵션 | 설명 |
| directory | 디렉토리 지정 (경로를 값에 작성) |
| name | 테스트 중 파일 이름 지정 |
| size | 측정에서 읽고 쓰는 파일 크기 |
| direct | 직접 또는 버퍼 I/O 설정, 값에 1입력시 버퍼 I/O를 사용 안함 |
| ioengine | 사용하는 I/O 엔진 변경하는 경우 사용 |
| iodepth | 테스트 중 동시에 발생하는 I/O 단위의 양을 설정 |
| numjobs | Job의 수 지정 |
| time\_based | runtime 만큼 I/O 실행 |
| runtime | 실행시간 지정 |
| group\_reporting | 결과를 job단위가 아니라 group단위로 출력 |
| norandommap | Random 워크로드 일 때, I/O 위치를 전에 위치와 상관없게 실행 |
| rw | I/O 종류 지정 (read, write, randread, randwrite) |
| bs | 테스트 중 사용 되는 블록 크기를 지정 |

## **3-3 실험 환경**

|  |  |
| --- | --- |
| 프로세서 | Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz |
| 커널버전 | 5.3.0-46-generic |
| 운영체제 | Ubuntu 18.04.3 LTS |
| 파일시스템 | BtrFS, Ext4 |
| 벤치마크 | Fio 3.1 |
| RAM | 32GB |
| SSD | Samsung SSD 860 PRO 256GB |

## **3-4 실험**

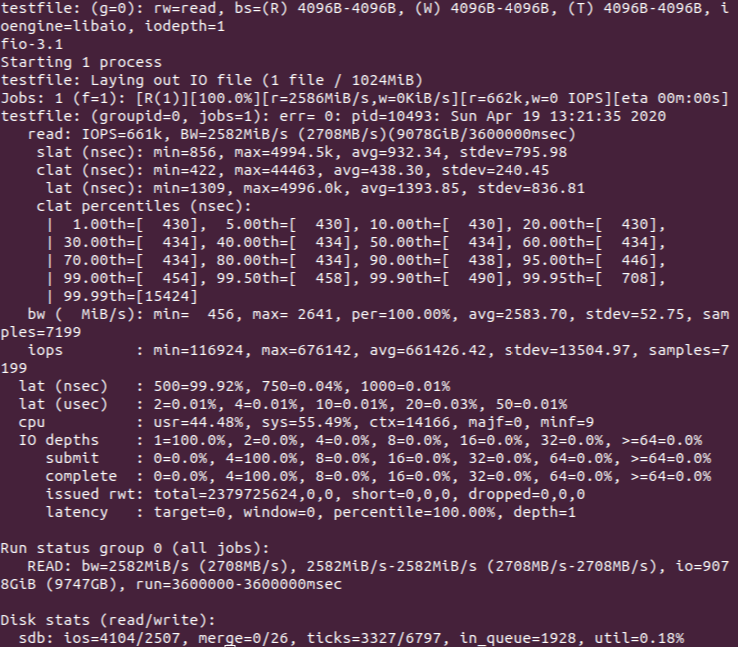
1. sudo fio 명령어 작성



[그림1] sudo fio 측정 명령어 작성

directory 값에는 경로를 작성하고 name 값은 생성되는 파일 이름을 작성한다. size는 이번 실험에서는 1G로 지정하였다. ioengine은 libaio를 설정하고 runtime은 3600으로 지정했다. rw는 순차읽기의 read, 순차쓰기 write, 랜덤읽기 randread, 랜덤쓰기 randwrite 중 필요한 옵션을 골라 넣습니다. bs는 4k 부터 64k를 넣으면서 실험합니다.

1. 실행결과 확인



[그림2] 실행 결과

1. rw 와 bs를 변경하며 반복하기

rw와 bs를 read, write, randread, randwrite, 4k, 8k, 16k, 32k, 64k를 바꿔가며 명령어를 실행시켜 IOPS와 BW의 값을 확인합니다 (이때, 입력 값을 바꾸기 전 file system을 포맷, 3-1 6번~7번 반복)

## **3-5 실험 결과**

순차 읽기에서는 IOPS와 BW가 두 파일시스템에서의 차이가 거의 없었다. IOPS는 두 파일시스템 다 블록사이즈를 증가시킬 때마다 작아졌고 BW는 블록사이즈를 증가시킬 때마다 커졌다.

순차 쓰기에서도 두 파일시스템은 블록사이즈 증가시킬 때마다 IOPS값이 작아졌고 BW의 값은 블록사이즈 증가 시 점점 늘어났다. 이때 IOPS값과 BW값의 차이가 같았다. 4k는 Ext4가 IOPS, BW값이 약1.19배 더 컸고 8k는 거의 차이가 없었으며 16k부터는 btrfs가 약1.19배, 32k는 약1.43배, 64k는 약1.61배 더 큰 값을 가졌다.

랜덤 읽기에서는 순차 읽기와 같이 서로의 차이는 거의 나지 않으며 IOPS는 블록사이즈가 커질수록 작아진다. BW는 블록사이즈가 커질수록 커진다.

랜덤 쓰기에서는 다른 읽기, 쓰기처럼 블록사이즈를 키울 때마다 IOPS값을 작아졌고 BW값은 커졌다. 또한 순차 쓰기처럼 ext4값이 4k일 때 약1.38배, 8k일 때는 약1.22배 컸지만 블록사이즈를 키울 때마다 차이가 좁혀지더니 16k에서는 거의 차이가 미미했다. 그리고 32k에서부터 차이가 약1.13배 생기며 btrfs가 더 커지고 64k에서는 약1.24배 차이가 났다.