CS344, Assignment 4 (Comparing filesystems)

Group Number 17

Name	Roll Number
1. Samiksha Sachdeva	180123040
2. Kartikay Goel	180101033
3. Falak Chhikara	180101023
4. Rahul Choudhary	180101060

We have chosen ZFS and ext4 file systems in order to quantitatively compare and quantify the benefits of the different new features in modern file systems.

The ZFS (Zettabyte File System)-:

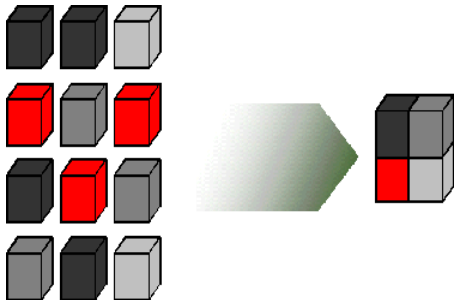
The ZFS is a revolutionary new file system having features and benefits that are not found in any other file system available today.

Pooled Data Storage-: ZFS uses the concept of **storage pools** to manage physical storage. ZFS **eliminates volume management** altogether**.**The concept of a **volume manager** was introduced to provide a representation of a single device so that file systems need not be modified to take advantage of multiple devices. This design prevented certain file system advances because the file system had no control over the physical placement of data on the virtualized volumes. Instead of forcing you to create virtualized volumes, ZFS aggregates devices into a storage pool. The storage pool describes the physical characteristics of the storage and acts as an arbitrary data store from which file systems can be created. **File systems are no longer constrained to individual devices, allowing them to share disk space with all file systems in the pool.** We no longer need to predetermine the size of a file system, as file systems grow automatically within the disk space allocated to the storage pool. When new storage is added, all file systems within the pool can immediately use the additional disk space without additional work. The storage pool works similarly to a virtual memory system. All processes on the system automatically use the additional memory.

Transactional Nature-: ZFS is a transactional file system, which means that the file system state is always consistent on disk. The problem of inconsistent file systems caused great pain to administrators. In ZFS, data is managed using **copy on write** semantics. Data is never overwritten, and any sequence of operations is either entirely committed or entirely ignored. Thus, the file system can never be corrupted through accidental loss of power or a system crash. The file system uses a 256-bit checksum, which is stored as metadata separate from the data it relates to, when it writes information to disk. Unlike a simple disk block checksum, this can detect phantom writes, misdirected reads and writes, DMA parity errors and driver bugs.

Simple, Efficient Administration-: Using ZFS commands, you can administer a system with short, efficient commands.

Deduplication-: If ZFS has the dedup property enabled, duplicate data blocks are removed as they are written to disk. The result is that only unique data is stored on disk and common components are shared between files as shown in the adjacent figure. In some cases, deduplication can result in savings in disk space usage and cost. However, you must consider the memory requirements before enabling the dedup property.



Compression-: It compresses your files on the fly and therefore lets you store more data using limited storage. At any time you can request ZFS compression stats per ZFS pool or volume and it will show you exactly how much space you're saving. Obviously, the biggest benefit of ZFS compression is that you can save quite a bit of space.

Here are some of the files that will be perfect for the ZFS compression scenario:
1. ISO images for new OS releases
2. VM images for VirtualBox
3. Docker container images
4. Large volumes of application or server logs

The ext4 File System-:

The **ext4** is a **journaling file system** for Linux, developed as the successor to ext3. **ext4** modifies important data structures of the filesystem such as the ones destined to store the file data. The result is a filesystem with an improved design, better performance, reliability, and features.

To meet various mission-critical requirements, the filesystem timestamps were improved with the addition of intervals down to nanoseconds. In **ext4**, **data allocation was changed from fixed blocks to extents.** An extent is described by its starting and ending place on the hard drive. This makes it possible to describe very long, physically contiguous files in a single inode pointer entry, which can significantly reduce the number of pointers required to describe the location of all the data in larger files. **ext4 reduces fragmentation** by scattering newly created files across the disk so that they are not bunched up in one location at the beginning of the disk.

Some of the important features of ext4 file system are as follows-:

Large file system-: The ext4 filesystem can support volumes with sizes up to 1 exbibyte and single files with sizes up to 16 tebibytes with the standard 4 KB block size.
Extents-: Extents replace the traditional block mapping scheme used by ext2 and ext3. An extent is a range of contiguous physical blocks, improving large-file performance and reducing fragmentation.
Backward compatibility-: ext4 is backward-compatible with ext3 and ext2, making it possible to mount ext3 and ext2 as ext4. This will slightly improve performance, because certain new features of the ext4 implementation can also be used with ext3 and ext2, such as the new block allocation algorithm, without affecting the on-disk format.

Persistent pre-allocation-: ext4 can pre-allocate on-disk space for a file. To do this on most file systems, zeroes would be written to the file when created. In ext4, `fallocate()`, a new system call in the Linux kernel, can be used. The allocated space would be guaranteed and likely contiguous. This situation has applications for media streaming and databases.

Delayed allocation-: ext4 uses a performance technique called allocate-on-flush, also known as *delayed allocation*. That is, ext4 delay block allocation until data is flushed to disk. Delayed allocation improves performance and reduces fragmentation by effectively allocating larger amounts of data at a time.

Journal checksums-: **ext4 uses checksums in the journal to improve reliability**, since the journal is one of the most used files of the disk. This feature has a side benefit: it can safely avoid a disk I/O wait during journaling, improving performance slightly.

Multiblock allocator-: When ext3 appends to a file, it calls the block allocator, once for each block. Consequently, if there are multiple concurrent writers, files can easily become fragmented on disk. However, ext4 uses delayed allocation, which allows it to buffer data and allocate groups of blocks. Consequently, the multiblock allocator can make better choices about allocating files contiguously on disk. The multiblock allocator can also be used when files are opened in O_DIRECT mode. This feature does not affect the disk format.

Comparison of ZFS and ext4-:

Feature	Present/Absent in ZFS	Present/Absent in ext4
Deduplication	Present	Absent
Compression	Present	Absent
Checksum	Present	Absent
Internal Snapshotting/Branching	Present	Absent
Persistent Cache	Present	Absent
Encryption	Present	Present
Block Journaling	Present	Present
Copy-on-write	Present	Absent

We have chosen the deduplication feature and the compression feature of ZFS. Both of these features are not present in ext4. So, we state the benefits of having these features by setting up a workload that helps to showcase these features. We run experiments using the same workload on both the filesystems. We pick some metrics in both sets of experiments and compare the value of those metrics in both the tests.

What is Deduplication?

Data deduplication is a technique for **eliminating duplicate copies** of repeating data. The main aim of deduplication is to go through large volumes of data and further identify large sections that are identical and finally, replacing them with a shared copy. During the process, unique chunks of data or byte patterns are first identified and then stored. Further, comparison of other chunks with the stored copy takes place. If there is a match, the repeated chunk is replaced with a small reference pointing to the stored chunk.

Storage-based data deduplication **reduces the amount of storage needed** for a given set of files. It is most efficient in case of applications with many copies of very similar data stored on a single disk. The most common form of data deduplication implementation compares chunks of data to detect duplicates. Each chunk of data is assigned an identification, generally using **cryptographic hash functions**. There exist other implementations too, which do not assume that two blocks of data with the same identifier are identical, and instead actually verify that data with the same identification is identical. Whatever be the implementation, it will further replace that duplicate chunk with a link.

Methods used for Deduplication-:

1. Chunking: In some systems, chunks are defined by physical layer constraints. The most intelligent (but CPU intensive) method to chunking is generally considered to be sliding-block. In the sliding block, a window is passed along the file stream to seek out more naturally occurring internal file boundaries.

2. Client backup deduplication: In this process, the deduplication hash calculations are initially created on the source (client) machines. Files that have identical hashes to files already in the target device are not sent, the target device just creates appropriate internal links to reference the duplicated data. The good thing about this method is that it avoids data being unnecessarily sent across the network, thereby reducing traffic load.

3. Primary storage and secondary storage: By definition, primary storage systems are designed for optimal performance, rather than lowest possible cost. The design criteria for these systems is to increase performance, at the expense of other considerations.

What is Compression?

Data compression is the process of **encoding information using fewer bits** than the original representation. A compression can be either lossy or lossless.

Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression.

Lossy compression reduces bits by removing unnecessary or less important information.

Compression is useful because it **reduces resources required to store and transmit data.** In the compression and decompression processes, computational resources are consumed. Data compression is subject to a space–time complexity trade-off. Factors like the degree of compression, the amount of distortion introduced (when using lossy data compression), and the computational resources required to compress and decompress the data, must be taken care of while designing data compression schemes.

Implementation of Deduplication feature in ZFS-:

Deduplication is used to save disk along with compression. There are three types of deduplication: file, byte and block. **ZFS** provides **block-level deduplication** for the very obvious reason that this is the finest granularity that makes sense for a general-purpose storage system.

File deduplication: In this type, each file is hashed with a cryptographic hashing algorithm. If the hash matches for multiple files, rather than storing the new file on disk, we reference the original file in the metadata. This type is the **most performant and least costly** on system resources. The drawback of this type is that if a single byte changes in the file, the hashes will no longer match. As a result, we can no longer reference the whole file in the filesystem metadata. So, we must make a copy of all the blocks to disk. This has **massive performance impacts in case of large files.**

Byte deduplication: This method is the **most expensive**, because we need to keep anchor points to determine where regions of deduplicated and unique bytes start and end. This type of deduplication works well for storage where a file may be stored multiple times, even if it's not aligned under the same blocks, such as mail attachments.

Block deduplication: ZFS uses block deduplication only. Block deduplication shares all the same blocks in a file, except the blocks that are different. As a result, we store only the unique blocks on disk, and reference the shared blocks in RAM. It's **more efficient than byte** deduplication, and **more flexible than file** deduplication. But this too has a drawback that it **requires a great amount of memory** to keep track of which blocks are shared, and which are not. However, because filesystems read and write data in block segments, it makes the most sense to use block deduplication for a modern filesystem.

Implementation of Compression on ZFS-:

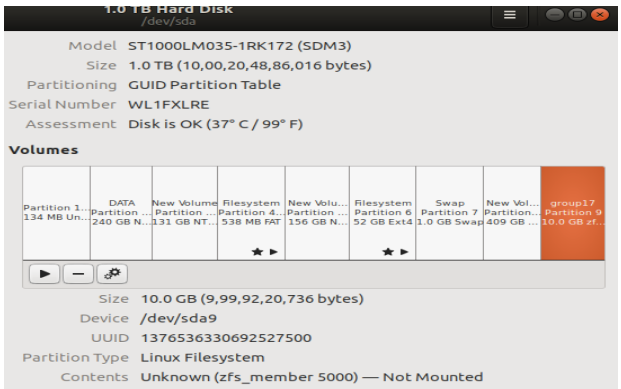
We have set the compression algorithm to LZ4 to quantify the benefits and disadvantages of the compression feature in ZFS.

LZ4 is a **lossless data compression** algorithm focused on compression and decompression speed. The algorithm gives a slightly worse compression ratio than the LZO algorithm. LZ4 only uses a dictionary-matching stage, and does not combine it with an entropy coding stage. (unlike other common compression algorithms)

This algorithm represents the data as a series of sequences. Each sequence begins with a one-byte token that is broken into two 4-bit fields. The first field represents the number of literal bytes that are to be copied to the output. The second field represents the number of bytes to copy from the already decoded output buffer (with 0 representing the minimum match length of 4 bytes). A value of 15 in either of the bitfields indicates that the length is larger and there is an extra byte of data that is to be added to the length. A value of 255 in these extra bytes indicates yet another byte to be added. Hence arbitrary lengths are represented by a series of extra bytes containing the value 255. The string of literals comes after the token and any extra bytes needed to indicate string length. This is followed by an offset that indicates how far back in the output buffer to begin copying. The extra bytes (if any) of the match-length come at the end of the sequence.

Compression can be carried out in a stream or in blocks. **Higher compression ratios** can be achieved by investing more effort in finding the best matches. This results in both a **smaller output and faster decompression**.

Installation of ZFS-:



We installed ZFS on a 10 GB partition(/dev/sda9) made in hard disk.

Step 1: Run the following command-:

sudo apt-get install zfsutils-linux

Step 2: Created a storage pool in the partition using a following command-:

sudo zpool create /dev/sda9 group17

Step 3: It lists all the pools in the ZFS file system.

sudo zpool list

```
kartikay@kartikay-goel:~$ zpool list
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
group17  9.25G   327M  8.93G         -     16%     3%  1.00x  ONLINE  -
```

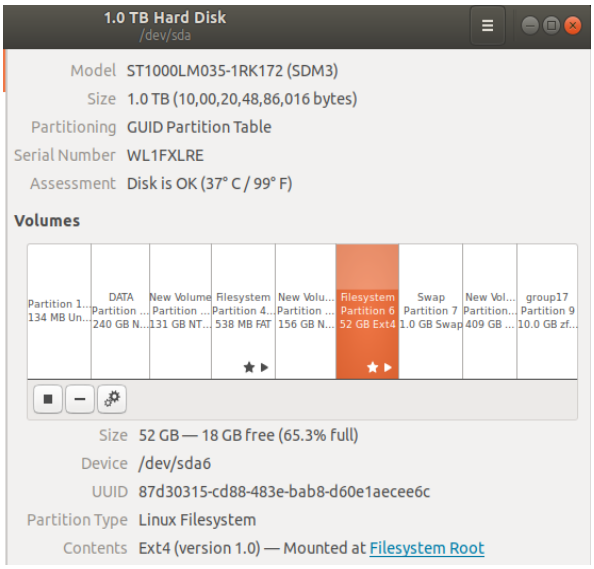
The **group17** ZFS pool should be mounted on /group17 automatically as we can see from the output of the df command.

```
kartikay@kartikay-goel:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            7.8G     0  7.8G   0% /dev
tmpfs           1.6G  2.2M  1.6G   1% /run
/dev/sda6        48G   31G   15G  69% /
tmpfs           7.8G  382M  7.4G   5% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           7.8G     0  7.8G   0% /sys/fs/cgroup
/dev/loop1      256M  256M     0 100% /snap/gnome-3-34
/dev/loop2      163M  163M     0 100% /snap/gnome-3-28
/dev/loop0      2.5M  2.5M     0 100% /snap/gnome-calc
/dev/loop7       98M   98M     0 100% /snap/core/10185
/dev/loop8       56M   56M     0 100% /snap/core18/188
/dev/loop9      141M  141M     0 100% /snap/gnome-3-26
/dev/loop10      63M   63M     0 100% /snap/gtk-common
/dev/loop4       45M   45M     0 100% /snap/gtk-common
/dev/loop5      1.0M  1.0M     0 100% /snap/gnome-logs
/dev/loop6      232M  232M     0 100% /snap/wine-platf
/dev/loop11     2.5M  2.5M     0 100% /snap/gnome-calc
/dev/loop3       98M   98M     0 100% /snap/core/10126
/dev/loop12     232M  232M     0 100% /snap/wine-platf
/dev/loop13     358M  358M     0 100% /snap/pycharm-ed
/dev/loop14     384K  384K     0 100% /snap/gnome-char
/dev/loop15     1.0M  1.0M     0 100% /snap/gnome-logs
/dev/loop17     3.8M  3.8M     0 100% /snap/gnome-syst
/dev/loop18     2.3M  2.3M     0 100% /snap/gnome-syst
/dev/loop22     141M  141M     0 100% /snap/gnome-3-26
/dev/loop23     162M  162M     0 100% /snap/gnome-3-28
/dev/loop16      74M   74M     0 100% /snap/wine-platf
/dev/loop20     384K  384K     0 100% /snap/gnome-char
/dev/loop19     218M  218M     0 100% /snap/gnome-3-34
/dev/loop21      56M   56M     0 100% /snap/core18/193
/dev/loop24     357M  357M     0 100% /snap/pycharm-ed
/dev/sda4       512M   24M  489M   5% /boot/efi
group17         9.0G  317M  8.7G   4% /group17
tmpfs           1.6G   16K  1.6G   1% /run/user/121
tmpfs           1.6G   68K  1.6G   1% /run/user/1000
```

Installation of ext4

ext4 is the default filesystem present in linux.

The **52 GB /sda/6 partition** shown in the above screenshot is the linux partition where the ext4 filesystem is installed by default.



Testing Deduplication Feature in ZFS:

To test the deduplication feature in ZFS, we created a workload using vdbench named example8. (attached in the zip file)

In this workload, we create 10 files of 20 mb each in a single directory.

```
*Example 8: Deduplication Feauture File system testing
dedupratio=3
dedupunit=20m
fsd=fsd1,anchor=/group17,depth=1,width=1,files=10,size=20m

fwd=fwd1,fsd=fsd1,operation=read,xfersize=4k,fileio=sequential,fileselect=random,threads=2

rd=rd1,fwd=fwd1,fwdrate=100,format=yes,elapsed=10,interval=1
```

We define the **File System Definition (FSD)** and use the following parameters:

1. **anchor** = /group17 and /home/kartikay/group17(for ZFS and ext4 respectively)
2. **depth** = 1(number of directories created in the parent directory)
3. **width** = 1 (number of directories created within directories)
4. **files**=10 (number of files created)
5. **size**=20m(size of each file is 20 mb)

The **filesystem workload definition (fwd)** has following parameters-:

1. **operation** = read (workload performs the read operation)
2. **Xfersize** = 4k(reads 4 blocks from selected file)
3. **Fileio** = sequential (sequentially reads blocks)
4. **Fileselect** = random (randomly selects files)
5. **threads**=2 (No. of threads to be used)

The **Run Definition (rd)** gas the following parameters:

1. **Format** = yes(the directory is completely formatted and initialized to our given structure)
2. **Elapsed** = 10(elapsed time for this defined run)
3. **Interval** =1(reporting interval)

The RD parameter **format=yes** causes the directory structure to be completely created, including initialization of all files to the requested size of **20m**. After the format completes the following will happen for 10(**elapsed=10**) seconds at a rate of 100 reads per second. It starts two threads**(threads=2)**. Each thread randomly selects a file **(fileselect=random)** and opens this file for read **(operation=read)**. It then sequentially reads 4k blocks **(xfersize=4k)** until end of file **(size=20m)**. Closes the file and randomly selects another file. **dedupunit=20m** denotes that the size of a data block that dedup tries to match with already existing data is 20 mb. **dedupratio=3** denotes that the ratio between the original data and the actually written data is 3:1

To test the above testcase, we run the following command-:

Step 1: First we set the dedup feature of ZFS to on using **sudo zfs set dedup=on group17**

Step 2: To run the test case, execute this**: sudo ./vdbench -f example8 -o test_deduplication_zfs.**

The above command creates a sub-directory in the same directory in which vdbench is present with the name test_deduplication_zfs. This directory contains html files which contains the results of the test**.** Most of the relevant information is present in the **summary.html** file.

The screenshot below shows the working of the vdbench test command.

```
kartikay@kartikay-goel:~$ zpool list
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
group17  9.25G  8.51M  9.24G         -    17%     0%  1.00x  ONLINE  -
kartikay@kartikay-goel:~$ sudo zfs set dedup=on group17
[sudo] password for kartikay:
kartikay@kartikay-goel:~$ cd Desktop/
kartikay@kartikay-goel:~/Desktop$ cd vdbench/
kartikay@kartikay-goel:~/Desktop/vdbench$ sudo ./vdbench -f example8 -o test_deduplication_zfs


Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.
Vdbench distribution: vdbench50407 Tue June 05  9:49:29 MDT 2018
For documentation, see 'vdbench.pdf'.

15:23:37.869 Created output directory '/home/kartikay/Desktop/vdbench/test_deduplication_zfs'
15:23:37.927 input argument scanned: '-fexample8'
15:23:37.927 input argument scanned: '-otest_deduplication_zfs'
15:23:37.983 Anchor size: anchor=/group17: dirs:            1; files:           10; bytes:   20
15:23:38.136 Starting slave: /home/kartikay/Desktop/vdbench/vdbench SlaveJvm -m localhost -n loc
15:23:38.765 All slaves are now connected
15:23:40.002 Starting RD=format_for_rd1
15:23:40.305 localhost-0: anchor=/group17 mkdir complete.

Nov 25, 2020 ..Interval.. .ReqstdOps... ...cpu%...  read ....read..... ....write.... ..mb/sec...
                          rate   resp total  sys   pct   rate   resp   rate   resp  read write
15:23:41.208            1  298.0 18.879  32.2 2.72   0.0    0.0  0.000  298.0 18.879  0.00 37.25
15:23:42.108            2  360.0 22.440  93.8 29.0   0.0    0.0  0.000  360.0 22.440  0.00 45.00
```

Ideally, 10 files of 20 mb size each should take 200 mb which combined with metadata takes approximately 209 mb. In the above case, the deduplication feature of ZFS was set to on and hence, we see that the deduplication is 3.33x and the space of memory used to store the files created is 69.7 mb which is (209/3), since we set dedupratio=3. The details are shown in the screenshot below.

```
15:23:56.248 READ_OPENS          Files opened for read activity:                 2          0/sec
15:23:56.248
15:23:56.466 Vdbench execution completed successfully. Output directory: /home/kartikay/Desktop/vdbench/test_deduplication_zfs

kartikay@kartikay-goel:~/Desktop/vdbench$ zpool list
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
group17  9.25G  69.7M  9.18G         -     17%     0%  3.33x  ONLINE  -
```

Now, we ran the same example with the dedup property=off. (**sudo zfs set dedup=off group17**)

Command-: **sudo ./vdbench -f example8 -o test_deduplicationoff_zfs**

```
kartikay@kartikay-goel:~/Desktop/vdbench$ sudo zfs set dedup=off group17
kartikay@kartikay-goel:~/Desktop/vdbench$ sudo ./vdbench -f example8 -o test_deduplicationoff_zfs


Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.
Vdbench distribution: vdbench50407 Tue June 05  9:49:29 MDT 2018
For documentation, see 'vdbench.pdf'.

15:30:58.079 Created output directory '/home/kartikay/Desktop/vdbench/test_deduplicationoff_zfs'
15:30:58.098 input argument scanned: '-fexample8'
15:30:58.098 input argument scanned: '-otest_deduplicationoff_zfs'
15:30:58.132 Anchor size: anchor=/group17: dirs:            1; files:           10; bytes:   200.0
15:30:58.163 Starting slave: /home/kartikay/Desktop/vdbench/vdbench SlaveJvm -m localhost -n local
15:30:58.365 All slaves are now connected
15:31:00.003 Starting RD=format_for_rd1
15:31:00.249 localhost-0: anchor=/group17 mkdir complete.

Nov 25, 2020 ..Interval.. .ReqstdOps... ...cpu%...  read ....read..... ....write.... ..mb/sec... m
                          rate   resp total  sys   pct   rate   resp   rate   resp  read write
15:31:01.178            1  301.0 20.568  29.5 2.44   0.0    0.0  0.000  301.0 20.568  0.00 37.62
15:31:02.053            2  331.0 23.005  89.5 26.1   0.0    0.0  0.000  331.0 23.005  0.00 41.38
15:31:03.036            3  384.0 20.879  78.3 5.03   0.0    0.0  0.000  384.0 20.879  0.00 48.00
```

Same number of files with the same size were created and now the space of memory that is used is 210 mb which is the total size of the files. This is because the deduplication property of ZFS was disabled during this experiment.

```
kartikay@kartikay-goel:~/Desktop/vdbench$ zpool list
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
group17  9.25G   210M  9.04G         -     17%     2%  1.00x  ONLINE  -
kartikay@kartikay-goel:~/Desktop/vdbench$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            7.8G     0  7.8G   0% /dev
tmpfs           1.6G  2.2M  1.6G   1% /run
/dev/sda6        48G   31G   15G  69% /
tmpfs           7.8G   61M  7.8G   1% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           7.8G     0  7.8G   0% /sys/fs/cgroup
/dev/loop0       45M   45M     0 100% /snap/gtk-common-themes/1353
/dev/loop2       63M   63M     0 100% /snap/gtk-common-themes/1506
/dev/loop4      1.0M  1.0M     0 100% /snap/gnome-logs/100
/dev/loop5      2.3M  2.3M     0 100% /snap/gnome-system-monitor/148
/dev/loop7      141M  141M     0 100% /snap/gnome-3-26-1604/98
/dev/loop8      1.0M  1.0M     0 100% /snap/gnome-logs/81
/dev/loop6       98M   98M     0 100% /snap/core/10126
/dev/loop3      256M  256M     0 100% /snap/gnome-3-34-1804/36
/dev/loop1      232M  232M     0 100% /snap/wine-platform-runtime/191
/dev/loop9      384K  384K     0 100% /snap/gnome-characters/570
/dev/loop10     162M  162M     0 100% /snap/gnome-3-28-1804/128
/dev/loop11     358M  358M     0 100% /snap/pycharm-educational/31
/dev/loop14     3.8M  3.8M     0 100% /snap/gnome-system-monitor/111
/dev/loop15      56M   56M     0 100% /snap/core18/1885
/dev/loop16     141M  141M     0 100% /snap/gnome-3-26-1604/100
/dev/loop17      98M   98M     0 100% /snap/core/10185
/dev/loop23     232M  232M     0 100% /snap/wine-platform-runtime/188
/dev/loop13     2.5M  2.5M     0 100% /snap/gnome-calculator/748
/dev/loop22     2.5M  2.5M     0 100% /snap/gnome-calculator/826
/dev/loop12     218M  218M     0 100% /snap/gnome-3-34-1804/60
/dev/loop18     357M  357M     0 100% /snap/pycharm-educational/30
/dev/loop19     163M  163M     0 100% /snap/gnome-3-28-1804/145
/dev/loop20      74M   74M     0 100% /snap/wine-platform-3-stable/6
/dev/loop21     384K  384K     0 100% /snap/gnome-characters/550
/dev/loop24      56M   56M     0 100% /snap/core18/1932
/dev/sda4       512M   24M  489M   5% /boot/efi
group17         9.0G  201M  8.8G   3% /group17
tmpfs           1.6G   16K  1.6G   1% /run/user/121
tmpfs           1.6G   56K  1.6G   1% /run/user/1000
```

Testing Deduplication Feature in ext4:

Now, we tested the deduplication feature in ext4 file system and we found that there was not even a slight deduplication. This shows that the deduplication property is absent in ext4.

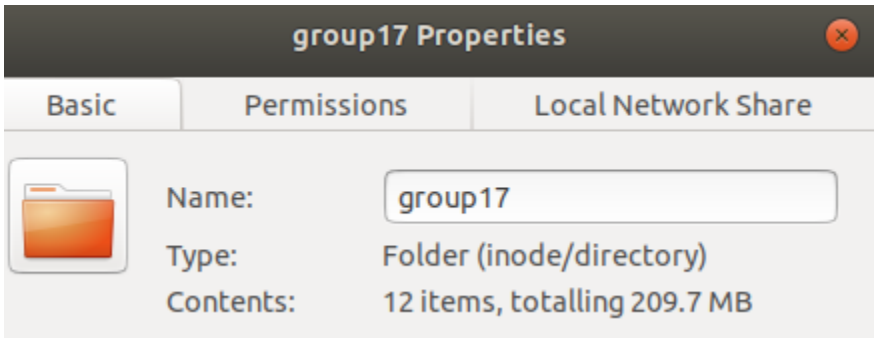
We ran example9 (same as example8 but the only difference is in the **anchor value**). We have set the **anchor= /home/kartikay/group17**. It creates the group17 directory in the specified anchor path.

```
kartikay@kartikay-goel:~/Desktop/vdbench$ sudo ./vdbench -f example9 -o test_deduplication_ext4


Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.
Vdbench distribution: vdbench50407 Tue June 05  9:49:29 MDT 2018
For documentation, see 'vdbench.pdf'.

15:35:46.221 Created output directory '/home/kartikay/Desktop/vdbench/test_deduplication_ext4'
15:35:46.240 input argument scanned: '-fexample9'
15:35:46.240 input argument scanned: '-otest_deduplication_ext4'
15:35:46.274 Anchor size: anchor=/home/kartikay/group17: dirs:            1; files:           10
15:35:46.302 Starting slave: /home/kartikay/Desktop/vdbench/vdbench SlaveJvm -m localhost -n loc
15:35:46.503 All slaves are now connected
15:35:46.538 localhost-0: Created anchor directory: /home/kartikay/group17
15:35:48.002 Starting RD=format_for_rd1
15:35:48.394 localhost-0: anchor=/home/kartikay/group17 mkdir complete.

Nov 25, 2020 ..Interval.. .ReqstdOps... ...cpu%...  read ....read..... ....write.... ..mb/sec...
                           rate   resp total  sys   pct   rate   resp   rate   resp  read write
15:35:49.209            1  247.0 19.738  27.0 2.45   0.0    0.0  0.000  247.0 19.738  0.00 30.88
```

The size of the contents of the directory made is 201 MB which we checked using the **sudo du -sh /group17/**

```
15:36:04.252
15:36:04.715 Vdbench execution completed successfully. Output directory: /home/kartikay/Desktop/vdbench/test_deduplication_ext4

kartikay@kartikay-goel:~/Desktop/vdbench$ cd ..
kartikay@kartikay-goel:~/Desktop$ cd ..
kartikay@kartikay-goel:~$ ls
Desktop  Documents  Downloads  examples.desktop  group17  Music  ns3  Pictures  Public  PycharmProjects  snap  Templates  Videos
kartikay@kartikay-goel:~$ sudo du -sh /group17/
201M    /group17/
```

The size of the directory is 209.7 MB which is different from 201 MB because the directory also contains some hidden files.



Quantitative Comparison of ZFS and ext4 filesystems on the basis of deduplication feature:

CPU Usage: If we have a look at the CPU usage, we can observe that the workload execution with **dedup on is much more CPU intensive** (22.3%) and the execution of same workload with **dedup off (2.89%)** is relatively much **lesser**. This also hints at the fact that in scenarios where there are no duplicate data, it is better to use dedup off than dedup on.

dedup = on(ZFS)

```
15:23:40.002 Starting RD=format_for_rd1

Nov 25, 2020 ..Interval.. .ReqstdOps... ...cpu%...
                             rate   resp total  sys
15:23:41.203            1  298.0 18.879  32.2 2.72
15:23:42.107            2  360.0 22.440  93.8 29.0
15:23:43.048            3  368.0 21.091  79.2 7.01
15:23:44.011            4  331.0 20.128  73.4 24.9
15:23:45.022            5  243.0  6.438  46.3 27.8
15:23:45.047      avg_2-5  325.5 18.485  73.2 22.3
```

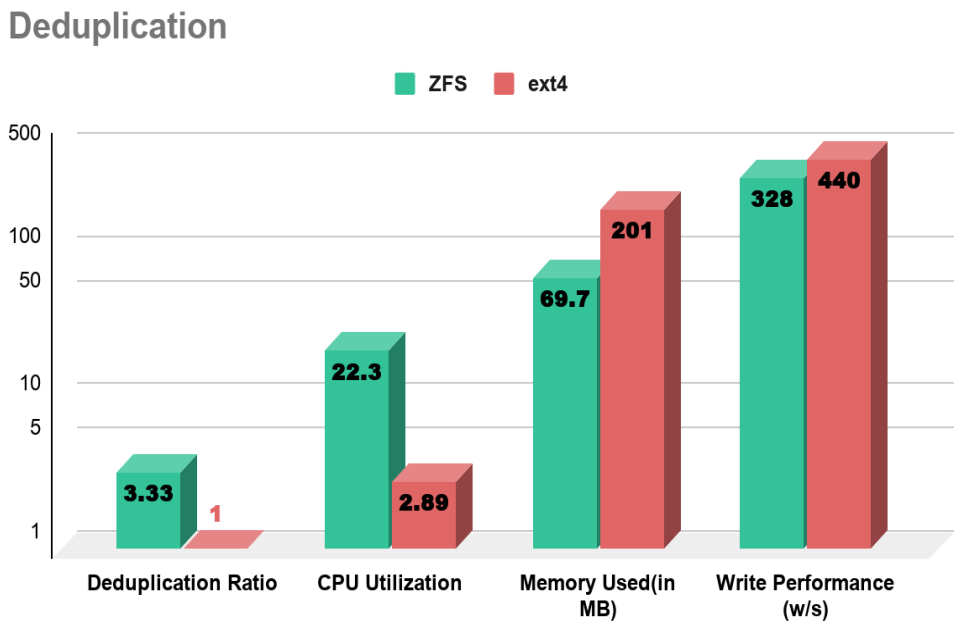
dedup=off (ext4)

```
15:35:48.002 Starting RD=format_for_rd1

Nov 25, 2020 ..Interval.. .ReqstdOps... ...cpu%...
                             rate   resp total  sys
15:35:49.201            1  247.0 19.738  27.0 2.45
15:35:50.054            2  370.0 21.207  83.5 3.43
15:35:51.038            3  411.0 19.432  80.0 3.77
15:35:52.012            4  395.0 13.936  57.6 2.68
15:35:53.025            5  177.0  5.574  15.6 1.73
15:35:53.052      avg_2-5  338.2 16.500  58.9 2.89
```

We can see that in the left image, the *CPU% total average* is 73.2% and *CPU% system average* is 22.3%. And in the right image for ext4, the *CPU% total average* is 58.9% and *CPU% system average* is 2.89%.

Memory Usage: With **dedup=on**, we observe that the time for **workload execution is less** and we also observe that the actual space taken is much less than the logical space needed. In contrast with **dedup=off**, where the **workload execution time is much more** as actual disk space needs to be provided even for the duplicated data and we can observe that the actual space taken is comparable to the logical space needed.

Write Performance: In the case of **ZFS**, when **dedup=on**, the **write rate is 325 writes/second** and in the case of **ext4**, the **write rate is 440 writes/second**. So, there is a significant difference in the write performance when the dedup feature is on, since whenever duplicates are found, we have to check the links to shared references in case of ZFS which is not the case in ext4.



Testing Compression Feature in ZFS:

To test the compression feature in ZFS, we created a workload using vdbench named example10(attached in the zip file).

In this workload, we create 7 files of 500 mb each in a single directory.

The RD parameter **format=yes** causes the directory structure to be completely created, including initialization of all files to the requested size of **500m**. After the format completes the following will happen for 30(**elapsed=30**) seconds at a rate of 100 reads per second. It starts

two threads**(threads=2)**. Each thread randomly selects a file **(fileselect=random)** and opens this file for read **(operation=read)**. It then sequentially reads 4k blocks **(xfersize=4k)** until end of file **(size=500m)**. Closes the file and randomly selects another file. **compratio=10 generates a data pattern that results in a 10:1 ratio.**

As there are different compression algorithms this parameter cannot guarantee the ultimate results. The data patterns implemented are based on the use of the 'LZJB' compression algorithm using a ZFS record size of 128k, and with that in mind the accuracy for this implementation is plus or minus 5%.

We use the LZ4 compression algorithm of ZFS to test our workload.

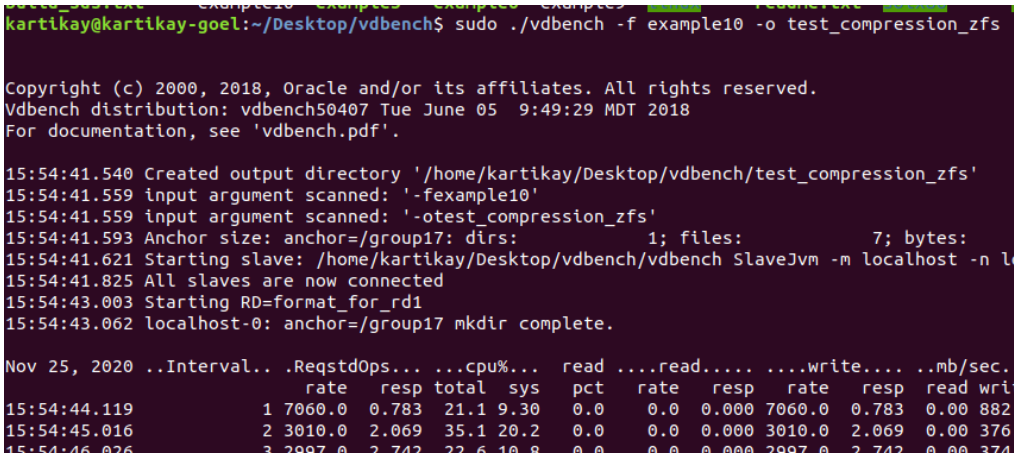
To test the above testcase, we run the following command-:

Step 1: First we set the compression algorithm of the compression feature of ZFS to LZ4 using **sudo zfs set compression=lz4 group17**

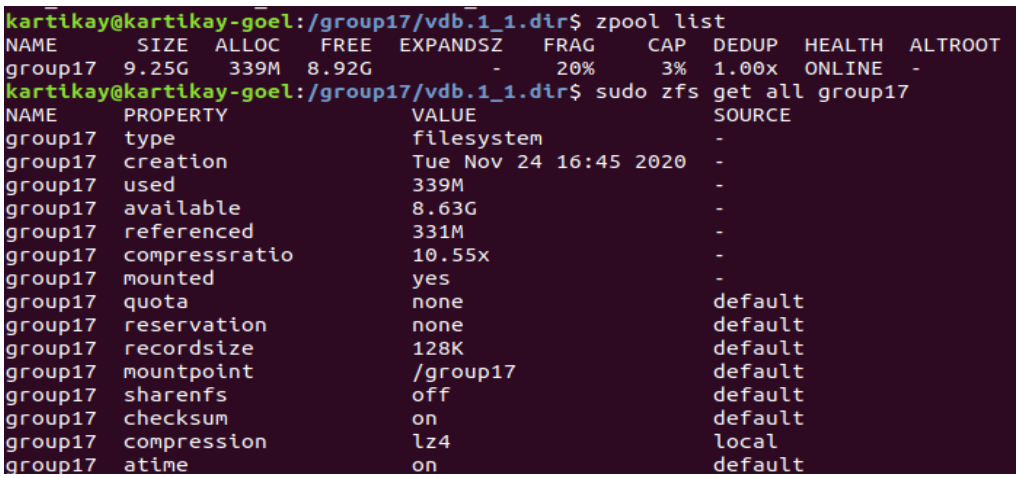
Step 2: To run the test case, execute this**: sudo ./vdbench -f example10 -o test_compression_zfs.**

The above command creates a sub-directory in the same directory in which vdbench is present with the name test_compression_zfs. This directory contains html files which contains the results of the test**.** Most of the relevant information is present in the **summary.html** file.

The screenshot below shows the working of the vdbench test command.



Ideally, 7 files of 500 mb size each should take 3.5gb which combined with metadata takes approximately 3.7gb. In the above case, the compression algorithm was set to LZ4. We see that the compression ratio is coming out to be 10.55x(shown in the screenshot below using **sudo zfs get all group17** command) and the space of memory used to store the files created is 339 mb(reported through the **zpool list** command) which is nearly (3700/10.55), since we set compratio=10. The details are shown in the screenshot below.



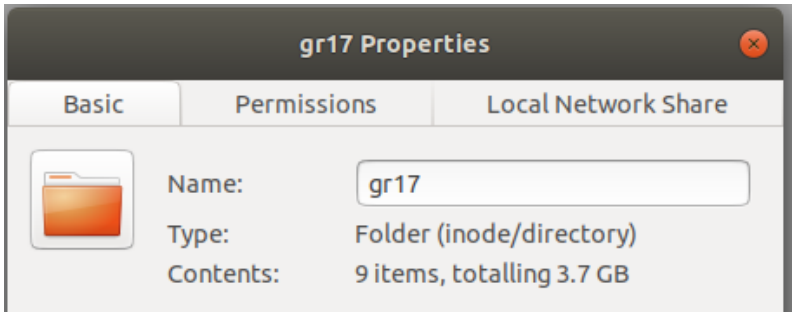
Testing Compression Feature in ext4:

Now, we tested the compression feature in ext4 file system and we found that there was not even a slight compression. This shows that the compression property is absent in ext4.

We ran example11(same as example10 but the only difference is in the **anchor** value). We have set the **anchor= /home/kartikay/group17**. It creates the **group17** directory in the specified anchor path.

```
kartikay@kartikay-goel:~/Desktop/vdbench$ sudo ./vdbench -f example11 -o test_compression_ext4


Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.
Vdbench distribution: vdbench50407 Tue June 05  9:49:29 MDT 2018
For documentation, see 'vdbench.pdf'.

16:37:00.947 Created output directory '/home/kartikay/Desktop/vdbench/test_compression_ext4'
16:37:00.980 input argument scanned: '-fexample11'
16:37:00.980 input argument scanned: '-otest_compression_ext4'
16:37:01.019 Anchor size: anchor=/home/kartikay/gr17: dirs:           1; files:           7; by
16:37:01.051 Starting slave: /home/kartikay/Desktop/vdbench/vdbench SlaveJvm -m localhost -n loca
16:37:01.259 All slaves are now connected
16:37:01.295 localhost-0: Created anchor directory: /home/kartikay/gr17
16:37:02.002 Starting RD=format_for_rd1
16:37:02.040 localhost-0: anchor=/home/kartikay/gr17 mkdir complete.

Nov 25, 2020 ..Interval.. .ReqstdOps... ...cpu%...  read ....read..... ....write.... ..mb/sec... 
                           rate   resp total  sys   pct   rate   resp   rate   resp  read write
16:37:03.449            1  14290  0.657  28.6 12.8   0.0    0.0  0.000  14290  0.657  0.00  1786
16:37:04.038            2  868.0  4.752  17.5 3.72   0.0    0.0  0.000  868.0  4.752  0.00 108.3
16:37:05.035            3  967.0  7.257  13.4 2.86   0.0    0.0  0.000  967.0  7.257  0.00 121.0
```

The size of the contents of the directory made is 3.7gb which we checked manually.



Quantitative Comparison of ZFS and ext4 filesystems on the basis of compression feature-:

CPU Usage: If we have a look at the CPU usage, we can observe that the workload execution when compression has taken place is much more CPU intensive(16.1%) and the execution of same workload with no compression(3.24%) is relatively much lesser. This is because the execution of LZ4 compression algorithm is CPU intensive and requires computational resources.

Compression in ZFS

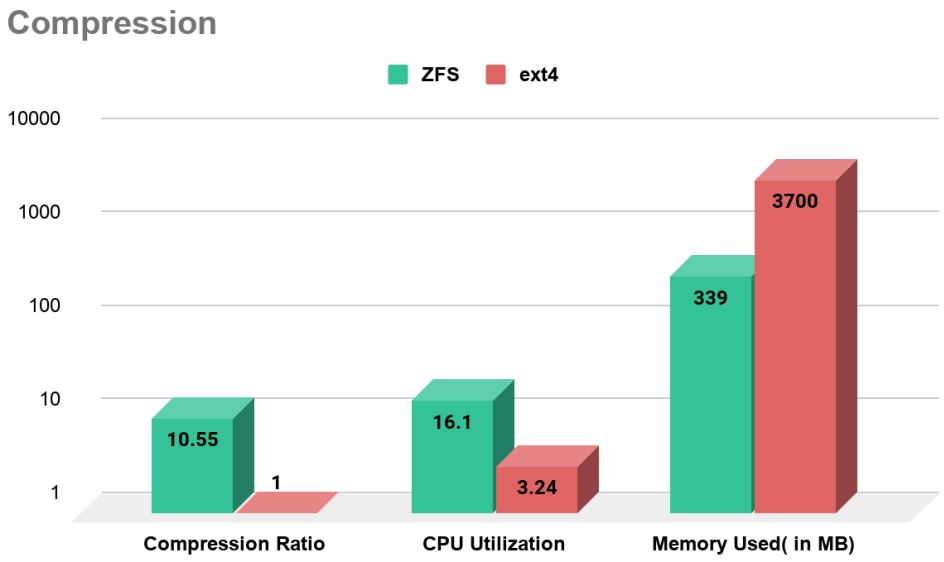
```
16:47:00.004 Starting RD=format_for_rd1

Nov 25, 2020 ..Interval.. .ReqstdOps... ...cpu%...
                           rate   resp total  sys
16:47:01.043            1 6569.0  0.782  25.5 15.2
16:47:02.045            2 3500.0  1.497  34.4 19.8
16:47:03.038            3 2997.0  2.505  25.7 14.4
16:47:04.028            4 3473.0  1.310  33.8 20.0
16:47:05.017            5 6905.0  1.698  41.3 24.2
16:47:06.033            6 2947.0  2.033  28.0 14.4
16:47:07.020            7    0.0  0.000  24.1 11.9
16:47:08.023            8 1609.0  4.833  18.1 8.10
16:47:08.035      avg_2-8 3061.6  1.996  29.3 16.1
```

Compression in ext4

```
Nov 25, 2020 ..Interval.. .ReqstdOps... ...cpu%...
                           rate   resp total  sys
16:37:33.008           31  189.0 10.072  17.1 3.05
16:37:34.013           32    0.0  0.000  18.5 3.19
16:37:34.030     avg_2-32  442.0 13.695  17.5 3.24
16:37:34.031     std_2-32  208.5 16.624
16:37:34.032     max_2-32  967.0 496.51
```

We can see that in the left image, the *CPU% total average* is 29.3% and *CPU% system average* is 16.1%. And in the right image for ext4, the *CPU% total average* is 17.2% and *CPU% system average* is 3.24%.

Memory Usage: With compression is there, we observe that the time for workload execution is less and we also observe that the actual space taken is much less than the logical space needed. In contrast with no compression case, where the workload execution time is much more and we can observe that the actual space taken is comparable to the logical space needed. The memory usage is less in case of ZFS due to LZ4 compression algorithm described above.



Are deduplication and compression always beneficial?

Is it always feasible to have dedup=on as we have seen that using this feature, memory used is very less?

The answer to the above question is that it is not always necessary to have dedup=on since it all depends on the type of data. If our data doesn't contain any duplicates, enabling dedup will add overhead without providing any benefit. If your data does contain duplicates, enabling dedup will both save space and increase performance. The space savings are obvious; the performance improvement is due to the elimination of disk writes when storing duplicate data, plus the reduced memory footprint due to many applications sharing the same pages of memory.

The main disadvantage of data compression is the performance impact resulting from the use of CPU and memory resources to compress the data and perform decompression.

Many vendors have designed their systems to try to minimize the impact of the processor-intensive calculations associated with compression. If the compression runs inline, before the data is written to disk, the system may offload compression to preserve system resources. If data is compressed after it is written to disk, or post-process, the compression may run in the background to reduce the performance impact. Although post-process compression can reduce the response time for each input/output (I/O), it still consumes memory and processor cycles and can affect the overall number of I/Os a storage system can handle. Also, because data initially must be written to disk or flash drives in an uncompressed form, the physical storage savings are not as great as they are with inline compression.